Michel Kleinschmidt

Automatisierte Generierung von Mappings zwischen Geschäftsobjekten, Präsentationsschicht und Geschäftslogik

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Mathematik / Physik / Informatik

Mittweida, 2009

Michel Kleinschmidt

Automatisierte Generierung von Mappings zwischen Geschäftsobjekten, Präsentationsschicht und Geschäftslogik

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Mathematik / Physik / Informatik

Geringswalde, 2009

Erstprüfer:	Prof. Dr. rer. nat. Konrad Schulz
Zweitprüfer:	Dipl.-Inf. (FH) André Friedrich
vorgelegte Arbeit wurde verteidigt am:	29.07.2009

Bibliographische Beschreibung:

Michel Kleinschmidt:
Automatisierte Generierung von Mappings zwischen Geschäftsobjekten, Präsentationsschicht und Geschäftslogik. - 2009. 62 S. Mittweida, Hochschule Mittweida, Fachbereich Mathematik/Physik/Informatik, Diplomarbeit, 2009

Referat:

Mappings werden überall dort eingesetzt, wo eine Zuordnung von Wertepaaren benötigt wird. Wenn dies oft und nach gleichen Mustern geschieht, dann ergibt sich eine Optimierungsmöglichkeit, die durch Automatisierung gelöst werden kann. Hier setzt die vorliegende Arbeit an. Es wird ein Konzept für die automatisierte Generierung von Mappings, in einem Softwareprojekt der Firma Innovations Software Technology GmbH, entwickelt. Anschließend wird das Konzept als Prototyp eines Mapping-Werkzeugs realisiert.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AM	Attribut-Mapping
API	Application Programming Interface
DOM	Document Object Model
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
EMFT	Eclipse Modeling Framework Technology
GEF	Graphical Editing Framework
J2SE	Java 2 Standard Edition
JAXP	Java API for XML Processing
JDOM	Java DOM
JEE	Java Enterprise Edition
JET	Java Emitter Templates
JSP	Java Server Pages
MVC	Model View Controller
OM	Objekt-Mapping
SAX	Simple API for XML
UML	Unified Modeling Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSLT	XSL Transformation

Abbildungsverzeichnis

Quelltextverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die vorliegende Diplomarbeit entstand im Auftrag der Innovations Software Technology GmbH, eines mittelständischen Unternehmens, dass hauptsächlich Software-Lösungen für Banken und Finanzdienstleister entwickelt. Derartige Anwendungen werden nach Erkenntnissen der modernen Software-Technik als Mehrschichten-Architekturen aufgebaut. Dabei spielen die Zuordnungen voneinander entsprechenden Elementen unterschiedlicher Schichten eine zentrale Rolle; sie werden mit dem technischen Begriff „Mappings“ bezeichnet (vgl. [LM07], S. 192).

Die vorliegende Arbeit thematisiert Mappings in einem speziellen Einsatzbereich zwischen Präsentationsschicht, Geschäftslogik und Geschäftsobjekten. Diese werden für die Zuordnung von Attributen der Präsentationsschicht zu den Attributen der Geschäftsobjekte verwendet. Der Einsatz von Mappings ist hier zweckmäßig, um die Anwendung nach dem Model-View-Controller-Entwurfsmuster gestalten zu können und dessen Vorteile zu nutzen.

Die Mappings werden im vorliegenden Fall immer nach ähnlichen Mustern erstellt. Das heißt, bestimmte Quelltextabschnitte sind immer gleich. Daraus ergibt sich die Möglichkeit, die Erstellung der Mappings zu automatisieren. Gleichzeitig wird durch die automatisierte Erstellung eine Reduzierung der Fehler während der Programmierung erreicht.

1.2 Zielsetzung

Das Ziel dieser Diplomarbeit ist die Konzeption und prototypische Realisierung eines Mapping-Werkzeugs, mit dem der Entwickler eine Beziehung zwischen einzelnen Attributen aus dem Geschäftsobjekt und der in der Benutzeroberfläche dargestellten Komponente herstellen kann. Aus diesen Beziehungen sollen dann automatisiert Mappings erstellt werden.

1.3 Aufbau der Diplomarbeit

Die vorliegende Diplomarbeit gliedert sich in folgende Bestandteile:

Kapitel 2 erklärt die grundlegenden Techniken und Begriffe, die für das Verständnis der Arbeit notwendig sind.

Kapitel 3 analysiert die aktuelle Situation der Mappings im Softwareprojekt und zeigt die Anforderungen an das zu entwickelnde Werkzeug.

Kapitel 4 zeigt Möglichkeiten für die Lösung der Aufgabenstellung auf, vergleicht diese und begründet die Entscheidung für einen bestimmten Lösungsansatz. Aufbauend auf dem gewählten Ansatz wird ein Konzept entwickelt.

Kapitel 5 beschreibt Details der Umsetzung für den aus Kapitel 4 gewählten Lösungsansatz. Dabei werden die interessantesten Gesichtspunkte der Implementierung erläutert und aufgezeigt.

Kapitel 6 fasst die Erkenntnisse und Leistungen der Arbeit zusammen.

Kapitel 2

Grundlagen und Begriffsklärung

2.1 Geschäftsobjekte und Geschäftslogik

Ein Geschäftsobjekt (engl. business object) stellt einen realen Gegenstand oder ein Konzept der Geschäftswelt dar. Es besitzt einen Namen, unter dem es in der Geschäftswelt bekannt ist, eine Bedeutung, einen Zweck, Eigenschaften, Verhalten, Beziehungen zu anderen Geschäftsobjekten, Regeln und Einschränkungen. Geschäftsobjekte sind also Objekte, die eine Bedeutung im Kontext der Geschäftstätigkeit des Unternehmens haben. Jedes Geschäftsobjekt ist, genauso wie seine Entsprechung in der realen Welt, eindeutig identifizierbar und in sich abgeschlossen. Daher können Geschäftsobjekte auch unabhängig voneinander entwickelt werden (vgl. [Oes06], S. 350 und [Jen99], S. 3 ff.). Ein typisches Beispiel für ein Geschäftsobjekt ist der Kunde. Er besitzt einen Namen, eine Adresse und weitere Eigenschaften. Es bestehen Beziehungen zu anderen Geschäftsobjekten, wie z.B. Kundenbetreuer oder Rechnung.

Unter Geschäftslogik (engl. business logic) versteht man programmierten Code, der die Funktionalität einer Anwendung umsetzt. Dazu gehören z.B. Geschäftsregeln, Datenvalidierung, Ablaufsteuerung und Sicherheit der Anwendung (vgl. [Lho06], S. 10 ff.). Der Begriff der Geschäftslogik hat auch eine zentrale Bedeutung im Zusammenhang mit mehrschichtigen Systemarchitekturen. Bei der Drei-Schicht-Architektur stellt die Geschäftslogik die Anwendungsschicht dar. Die Anwendungsschicht wird auch als Geschäftslogik- oder Applikationsschicht bezeichnet. (vgl. [DH03], S. 17 ff.)

2.2 Präsentationsschicht

Die Präsentationsschicht (engl. presentation layer) bezeichnet in mehrschichtigen Systemarchitekturen eine Darstellungsschicht, die eine Benutzeroberfläche zur Verfügung stellt, um Daten anzuzeigen und auf Benutzereingaben zu reagieren. Dabei werden die Daten und Informationen der Geschäftsobjekte in geeigneter Form auf der Benutzeroberfläche durch typische Darstellungsmittel wie Textfelder, Radio-Buttons, Checkboxen usw. dargestellt. Durch Aktionen von Benutzern, wie der Klick auf eine Schaltfläche, wird auf der Benutzeroberfläche ein Ereignis ausgelöst und an die darunterliegende Schicht weitergeleitet (vgl. [DH03], S. 18 ff.).

2.3 Model View Controller

Das Model-View-Controller-Entwurfsmuster unterteilt eine Anwendung in drei Komponenten. Das Modell stellt die Anwendungskomponente dar. Die Ansicht (engl. view) präsentiert dem Anwender Informationen. Die Steuerungskomponente (engl. controller) ist für die Verarbeitung der Bedieneingaben verantwortlich. Ansichten und Steuerungskomponenten bilden zusammen die Benutzerschnittstelle. Das Model-View-Controller-Entwurfsmuster ermöglicht die Entkopplung der drei genannten Komponenten einer Anwendung, wodurch die Flexibilität, Wartbarkeit und Wiederverwendbarkeit der Anwendung erhöht wird (vgl. [G02], S. 5).

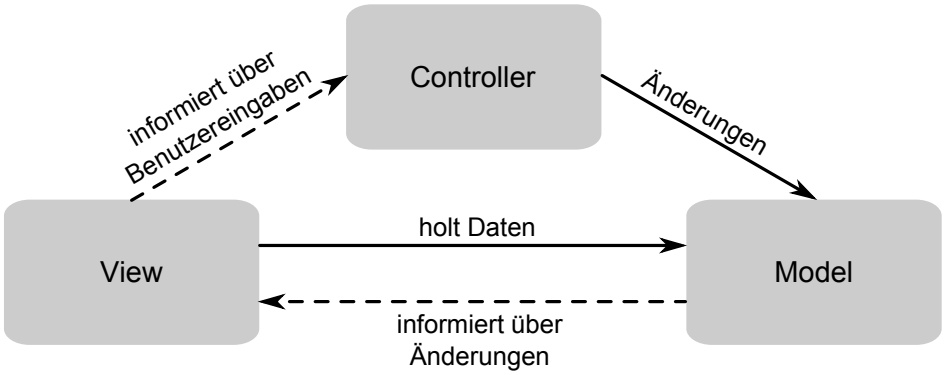

Abbildung 2.1: Model View Controller [MSH03, S. 255]

Die Entkopplung der Komponenten wird durch den Aufbau eines Protokolls zur Benachrichtigung erreicht. Eine Ansicht muss sicherstellen, dass seine Darstellung den Zustand des Modells wiedergibt. Das Modell benachrichtigt die von ihm abhängigen

Ansichten, wenn seine Daten sich ändern. Dadurch erhält eine Ansicht die Möglichkeit, sich selbst an die Änderung anzupassen. Dieser Ansatz ermöglicht die Bindung mehrerer Ansichten an ein Modell. So können auch neue Ansichten für ein Modell entwickelt werden, ohne Änderungen an diesem vornehmen zu müssen (vgl. [G02], S. 6). In der Abbildung 2.1 wird die grundlegende Funtkionsweise des Model View Controller (MVC)-Entwurfsmusters dargestellt.

2.4 Framework

Ein Framework (deutsche Übersetzung für Framework: Rahmenwerk) ist ein "teilweise fertiges“ Softwaresystem, bestehend aus einer Vielzahl von auf einander abgestimmten Softwarekomponenten, aus denen mit geringem Aufwand ein angepasstes Softwaresystem erstellt werden kann. Hierbei wird ein wieder verwendbares Design zur Verfügung gestellt, das aus vorgefertigten Komponenten und Regeln für die Interaktion dieser Komponenten besteht (vgl. [HN05a], S. 229).

2.5 Work Frame Relations

Work Frame Relations ist ein branchenspezifisches Framework der Firma Innovations Software Technology GmbH zur Entwicklung einer individuellen Bankenanwendung. Es ist eine Weiterentwicklung des Framework Work Frame und verwendet die Innovations-Regeltechnologie Visual Rules. Es stellt ein flexibles Geschäftsobjektmodell zur Abbildung von Geschäftsbeziehungen, Partnern, Beziehungen, Customer-Relationship-Management-Daten, Konten, Depots, Portfolios, Transaktionen und mehr zur Verfügung. Work Frame Relations enthält einen grafischen Relationen-Browser zur Darstellung von Personen, Geschäftsbeziehungen, Rollen und Verbindungen. Es beinhaltet vorgefertigte Komponenten wie Charts, Reports, Analysefunktionen, Tabellen und dynamische Kennzahlen. Work Frame Relations stellt regelgesteuerte Wizards zur einfachen Erfassung komplexer Informationen, z.B. Neueröffnung einer Geschäftsbeziehung, einer Person eines Kontaktes, zur Verfügung. Zum Funktionsumfang gehören weiterhin:

- elektronische Workflows
- mehrsprachige und durchgängige Mandantenfähigkeit
- Sicherheitsarchitektur und mehrstufige Benutzer/Rechte/Rollenverwaltung

- skalierbare Architektur mit durchgängigem Einsatz von Java Enterprise Edition (JEE), sowie Nutzung eines Application-Server
- Anpassungsfähigkeit an das Corporate Design von Auftraggebern

(vgl. [Inn08], S. 1 ff.)

2.6 Reflection API

Bei dem Reflection Application Programming Interface (API) handelt es sich um eine Java-Klassenbibliothek, die einem Programm zur Laufzeit den Zugriff auf Attribute und Methoden von geladenen Klassen erlaubt. Die entsprechenden Objekte können auch unter Beachtung oder Nichtbeachtung der Sichtbarkeitsbeschränkungen manipuliert werden (vgl. [Krü02], S. 989 ff.).

Weitere Möglichkeiten der Reflection API sind: das Erzeugen von Objekten und Arrays mit dynamischen Typen, das Auslesen von Annotations, dynamische Methodenaufrufe und auch das Abschalten von Zugriffsbeschränkungen (vgl. [Loc08], S. 1 ff.).

2.7 Mappings

Ein Mapping ist eine Zuordnung von Daten aus zwei verschiedenen Feldern, Speicherbereichen oder Protokollen zueinander (vgl. [DAT08]). Der Begriff Mapping stammt vom englischen *to map*, was auf deutsch *abbilden* bedeutet. Er wird in vielen technischen und wissenschaftlichen Gebieten verwendet. Unter anderem in der Mathematik oder der Grafik- und 3D-Modellierung. In dieser Arbeit werden unter dem Begriff Mappings immer Zuordnungen im Bereich der Informatik verstanden.

Kapitel 3

Analyse und Anforderungen

3.1 Ausgangslage

3.1.1 Einsatzgebiet und Aufgaben von Mappings

Der Einsatzort für die in der vorliegenden Arbeit thematisierten Mappings ist eine auf Work Frame Relations aufbauende Bankenanwendung für eine schweizer Privatbank. Die Kernaufgabe der Mappings besteht darin, die Attribute der Geschäftsobjekte den Elementen der jeweiligen Benutzeroberflächen zuzuordnen. Dafür werden sogenannte symbolische Bezeichner zur eindeutigen Indentifizierung der Benutzeroberflächenelemente verwendet. Diese werden in den Java-Klassen als String-Konstanten realisiert. Die symbolischen Bezeichner werden als „gemeinsame Sprache“ in den Eingabemasken der Benutzeroberfläche, in den Mappings und innerhalb der Geschäftslogik (Visual Rules Regelsystem), eingesetzt. Visual Rules ist eine Business Rules Management Plattform. Die Plattform ermöglicht ein effizientes Management von Geschäftsregeln, angefangen von der Regelerstellung, über Simulation, Test, Integration und Übergabe, bis hin zum Monitoring und der anschließenden Pflege. In Quelltext 3.1 sieht man ein Beispiel für die Verwendung des symbolischen Bezeichners `GBZ_ALLGEMEINE_BEMERKUNG` für ein Mapping.

Durch den Einsatz der Mappings kann innerhalb der Anwendung ein klares MVC-Muster realisiert werden. Dabei ermöglichen die Mappings eine Trennung zwischen Geschäftslogik und Präsentationsschicht. Außerdem erreicht man eine verbesserte Wartbarkeit der einzelnen Module, eine Auflösung der Abhängigkeiten zwischen den Eingabemasken der Benutzeroberfläche und eine verbesserte Performance.

Die Eingabemasken der Benutzeroberfläche werden mit Extensible Markup Language (XML)-Dateien realisiert. Dies wird durch den Einsatz eines firmeneigenen Frameworks möglich. Das Framework erlaubt es, die Elemente der Benutzeroberfläche mithilfe von XML-Tags zu realisieren. Zum Beispiel wird das Tag-Element `TextArea` vom Framework zur Laufzeit in das Swing-Objekt `JTextArea` umgewandelt. Die Eingabemasken können durch die XML-Notation schneller und übersichtlicher erstellt werden, da man sich allein auf die einzelnen Elemente der jeweiligen Eingabemaske konzentrieren kann. Dinge wie Layout-Manager und Listener müssen in den XML-Dateien nicht beachtet werden.

3.1.2 Mapping-Typen

Die Mappings unterteilen sich in zwei Arten: Objekt-Mappings und Attribut-Mappings. Für jedes Geschäftsobjekt existiert genau ein Objekt-Mapping (OM). Für das Geschäftsobjekt `Geschäftsbeziehung` gibt es zum Beispiel das OM `GBZMapping`. Zwischen dem OM und dem Attribut-Mapping (AM) besteht eine Aggregationsbeziehung. Die AMs werden innerhalb der OMs mithilfe von inneren Klassen realisiert. Quelltext 3.1 zeigt dafür ein Beispiel aus der Klasse `GBZMapping`.

```
1  private Mapping<String, Geschaeftsbeziehung> createAllgemeinBemerkung()
2  {
3     return new Mapping<String, Geschaeftsbeziehung>(
          GBZ_ALLGEMEINE_BEMERKUNG, String.class)
4     {
5        @Override
6        protected String getValue(Geschaeftsbeziehung businessObject,
             IMappingContext context)
7        {
8           return businessObject.getPotenziellerKundeBemerkung();
9        }
10
11       @Override
12       protected void setValue(String value, Geschaeftsbeziehung gbz,
             IMappingContext context)
13       {
14          gbz.setPotenziellerKundeBemerkung(value);
15       }
16    };
17 }
```

Quelltext 3.1: Attribut-Mapping als innere Klasse

Hier ist auch zu sehen, dass das AM über eine Methode an den Konstruktor des OMs geliefert wird. Im Konstruktor werden die AMs mit der Methode `add()` hinzugefügt. Für das Beispiel aus Quelltext 3.1 wäre die zugehörige Zeile im Konstruktor `add(createAllgemeineBemerkung());`.

Das OM definiert des Weiteren die bereits erwähnten symbolischen Bezeichner und eine Implementierung des OMs als Singleton-Pattern. Dadurch kann sichergestellt werden, dass von einem OM auch immer nur eine Instanz zur Laufzeit existiert.

Die Kernfunktionalität des AMs ist die Zuordnung eines Attributs des Geschäftsobjektes zu einem symbolischen Bezeichner, der mit den Elementen einer Eingabemaske verbunden ist. AMs gibt es in verschiedenen Typen. Dabei wurden vom allgemeinen AM-Typ weitere abgeleitet, um für spezielle Elemente der Benutzeroberfläche zusätzlich Methoden zur Verfügung zu stellen. Das Klassendiagramm in Abbildung 3.1 zeigt die verschiedenen AM-Typen und ihre Vererbungsstruktur.

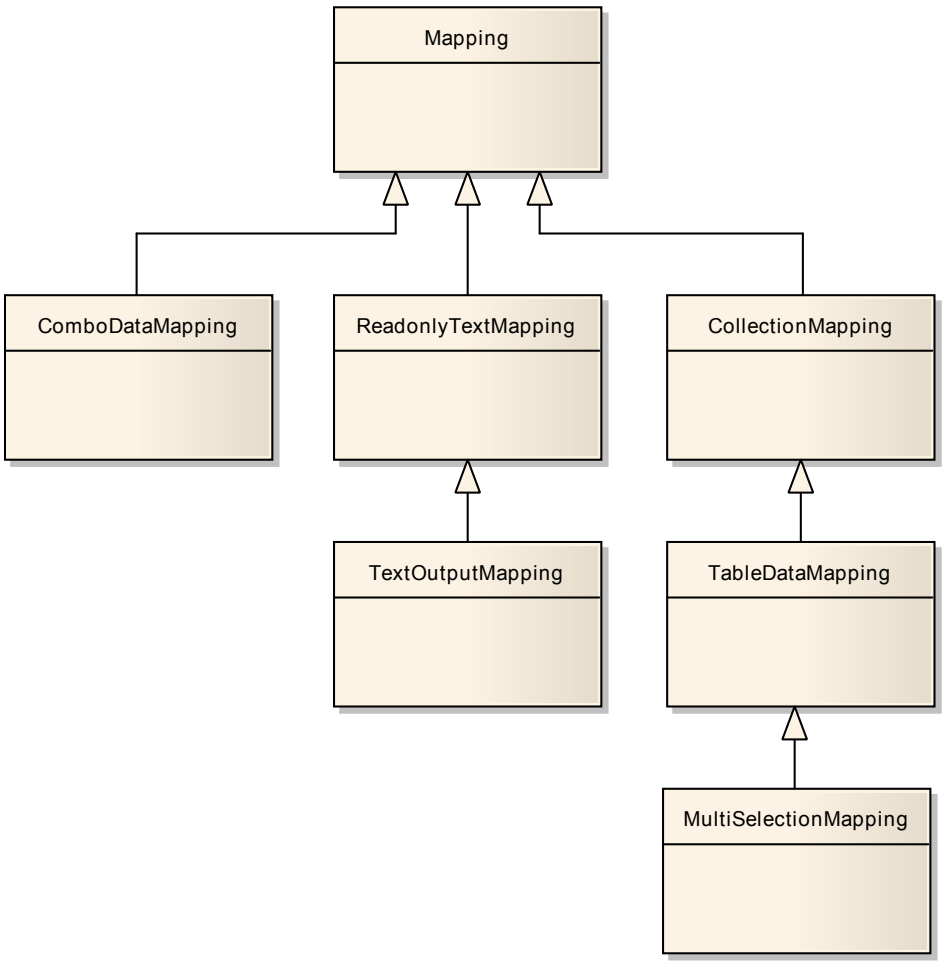

Abbildung 3.1: Attribut-Mapping-Typen

Zum Beispiel eignet sich der AM-Typ `ComboDataMapping` vor allem für die Zuordnung einer `ComboBox`. Das `ComboDataMapping` definiert die Methode `createTableData()`. Diese Methode gibt ein Objekt vom Typ `ITableData` zurück, welches für die Befüllung der `ComboBox` in der Eingabemaske benötigt wird.

3.2 Anforderungen

Um die Hauptziele der Zeitersparnis während der Mapping-Erstellung und der Fehlervermeidung bei deren Programmierung zu erreichen, sollen die folgenden Anforderungen an das Mapping-Werkzeug erfüllt werden.

Die automatisiert erstellten Mappings sollen wiederverwendbar sein. Einmal kofigurierte Zuordnungen zwischen bestimmten Geschäftsobjekten und Teilen der Benutzeroberfläche (XML-Eingabemasken) sollen auf Wunsch wiederverwendet, erweitert und verändert werden können. Das erstellte Mapping-Werkzeug soll erweiterbar sein. Beim Design der Benutzeroberfläche und der Benutzerführung des Mapping-Werkzeugs soll die Verwendung von zu vielen, möglicherweise ineinandergeschachtelten Wizards verhindert werden. Es soll hier eher eine programmier- und entwicklernahe Benutzerführung ermöglicht werden.

Kapitel 4

Entwurf

4.1 Grundsätzliches Vorgehen bei der automatischen Generierung

Aufbauend auf dem bisherigen System von OMs und AMs wird Entschieden, die Mappings durch Quelltextgenerierung und Erzeugung von Klassen automatisiert zu erstellen. Das hat den Vorteil, dass die generierten Klassen direkt in Jar-Dateien bzw. Klassenbibliotheken im Projekt weiterverwendet werden können.

Als nächstes muss geklärt werden, was überhaupt generiert werden kann. Man kann nur den Quelltext generieren, der in jedem Fall gebraucht wird. Das ist der kleinste gemeinsame Teil, der bei jeder Mapping-Implementierung bisher benötigt wurde. Jede Art von Spezialfall muss vom Benutzer immer noch selbst programmiert werden. Zunächst wurde mit der Betrachtung der AMs begonnen. Sie sind der elementare Teil der Mappings, da sie die Kernfunktionalität, der Zuordnung eines Geschäftsobjektattributs zu einem symbolischen Bezeichner eines Eingabemaskenelementes enthalten. Dabei wurde herausgefunden, dass in allen AMs die Methode `getValue` implementiert ist. Wenn es sich nicht um ein „Readonly-Attribut“ handelt, wird auch in jedem Fall die Methode `setValue` verwendet. In Quelltext 4.1 kann man ein Beispiel für eine Standardimplementierung der genannten Methoden sehen. Das dargestellte AM aus dem OM `GBZMapping` ist vom Typ `Mapping` abgeleitet.

```
1  private Mapping<String, Geschaeftsbeziehung> createAllgemeinBemerkung()
2  {
3    return new Mapping<String, Geschaeftsbeziehung>(
         GBZ_ALLGEMEINE_BEMERKUNG, String.class)
4    {
5       @Override
6       protected String getValue(Geschaeftsbeziehung businessObject,
            IMappingContext context)
7       {
8          return businessObject.getPotenziellerKundeBemerkung();
9       }
10
11      @Override
12      protected void setValue(String value, Geschaeftsbeziehung gbz,
            IMappingContext context)
13      {
14         gbz.setPotenziellerKundeBemerkung(value);
15      }
16   };
17 }
```

Quelltext 4.1: Beispiel Attribut-Mapping

Das gezeigte Beispiel ist die einfachste Implementierung eines AM. In den Methoden `getValue` und `setValue` wird nur die Get- und Set-Methode des Geschäftsobjektattributs aufgerufen. Eine solche Implementierung eines AM kann komplett generiert werden. Zusätzlich gibt es in allen von `Mapping` abgeleiteten AM-Typen neben den genannten Get und Set spezielle Methoden. Zum Beispiel die Methode `createTableData` vom AM-Typ `ComboDataMapping`, die wie am Ende von Abschnitt 3.1.2 bereits erwähnt, ein Objekt vom Typ `ITableData` zurück gibt, welches für die Befüllung der `ComboBox` in der Eingabemaske benötigt wird. Diese Methode wird in jedem `ComboDataMapping` verwendet. Ihre Implementierung ist jedoch immer unterschiedlich. Man kann die Methode trotzdem generieren. Allerdings wird sie abstrakt deklariert. Der Benutzer muss dadurch ihren Inhalt selbst implementieren, wenn er von der erstellten Klasse ableitet. Im Abschnitt 4.5 werden im Zusammenhang mit dem Systementwurf die Ausführungen dieses Abschnitts fortgesetzt.

4.2 Generierung des Quelltextes

Im vorhergehenden Abschnitt wurden grundsätzliche Überlegungen zu den Vorgehensweisen bei der Mapping-Generierung erörtert. In den folgenden Abschnitten werden nun Möglichkeiten und Techniken vorgestellt, die zur Generierung des Quelltextes verwendet werden können. Anschließend wird eine der Möglichkeiten ausgewählt und die Wahl begründet.

4.2.1 Eigener Quelltextgenerator

Die Entwicklung eines eigenen Quelltextgenerators bietet die Flexibilität, diesen genau für die gewünschten Umstände anzupassen. Das Gesamtkonzept des Systems muss durch die Einschränkungen des Quelltext-Generators nicht verändert werden. Es ist möglich ihn, als unabhängige externe Komponente oder direkt im Programm zu integrieren. Man kann den Generator speziell auf die Erstellung von Java-Quelltext zuschneiden. Dabei muss dieser bestimmte Anforderungen, wie die Erstellung beliebigen Java-Quelltextes, erfüllen.

Vorteile:

- Eine eigene Lösung eröffnet bei der Erstellung vollkommene Flexibilität.
- Die Lösung kann speziell an die Erfordernisse der Mapping-Quelltextgenerierung angepasst werden.

Nachteile:

- Der Zeitaufwand für die Entwicklung eines eigenen Konzepts kann schwer eingeschätzt werden.
- Das Rad wird unter Umständen neu erfunden.

4.2.2 EMF und JET

Das Eclipse Modeling Framework (EMF) ist ein komplexes Modellierungs- und Codegenerierungs-Framework. EMF erlaubt die Entwicklung von Java-Anwendungen auf der Basis von einfachen Modell-Definitionen. Die Modell-Definitionen können mithilfe von drei verschiedenen Techniken erstellt werden: Java, XML und Unified Modeling Language (UML). Dabei wählt sich der Benutzer eine Technik aus und erstellt mit deren Hilfe sein Modell. Dieses Modell, beispielsweise in Form von einem XML-Schema, kann bei Wunsch in Java-Interfaces oder UML-Diagramme umgewandelt werden. Dadurch hat der Benutzer die Freiheit sich seine Modellierungstechnik selbst zu wählen und sie bei Bedarf umzuwandeln. Aus den erstellten Modellen lässt sich mithilfe der Quelltextgenerierungsfunktionen des EMF jederzeit Quelltext erzeugen (vgl. [Bud03], S. 9 ff.).

Java Emitter Templates (JET) ist ein Quelltext-Erzeugungswerkzeug des EMF. Es bietet die Möglichkeit, Quelltext verschiedenster Art zu erzeugen. Von SQL, Java, XML bis zu normalem Text kann JET alles generieren (vgl. [SS05], S. 5).

JET ist als Teil von EMF für den Einsatz innerhalb von Eclipse vorgesehen. Für die Verwendung von JET benötigt man in der Regel eine Eclipse-Version mit einer kompatiblen EMF-Version. Die Verwendung von JET ohne Eclipse und EMF ist nicht möglich. Es gibt jedoch eine Variante, Eclipse im „Headless“-Modus (ohne Benutzeroberfläche) laufen zu lassen. Dadurch ist es möglich, JET über die Kommandozeile zu nutzen (vgl. [Pop04] und [SS05], S. 6).

Der Quelltext-Erzeugungsprozess (Abbildung 4.1) unterteilt sich in zwei Schritte:

1. Eine Vorlage (engl. template), die der späteren Ausgabe entspricht, wird erstellt. Diese wird beim Speichern automatisch von JET analysiert und in eine so genannte Implementierungsklasse übersetzt. Dieser Vorgang wird als Übersetzung (engl. translation) bezeichnet (vgl. [SS05], S. 6).

2. Von der erzeugten Implementierungsklasse können nun beliebig Objekte erzeugt werden. Um die gewünschte Ausgabe zu erhalten, muss nur die Methode `generate` des Exemplars aufegerufen werden. Dieser Vorgang wird als Generierung (engl. generation) bezeichnet (vgl. [SS05], S. 6).

JET verwendet eine Vorlagen-Technik (engl. template technology), die sich stark an die Syntax der Java Server Pages (JSP) anlehnt. Allerdings ist die Vorlagen-Technik mit der JSP-Syntax nicht identisch. Die Syntax definiert drei verschiedene Arten von

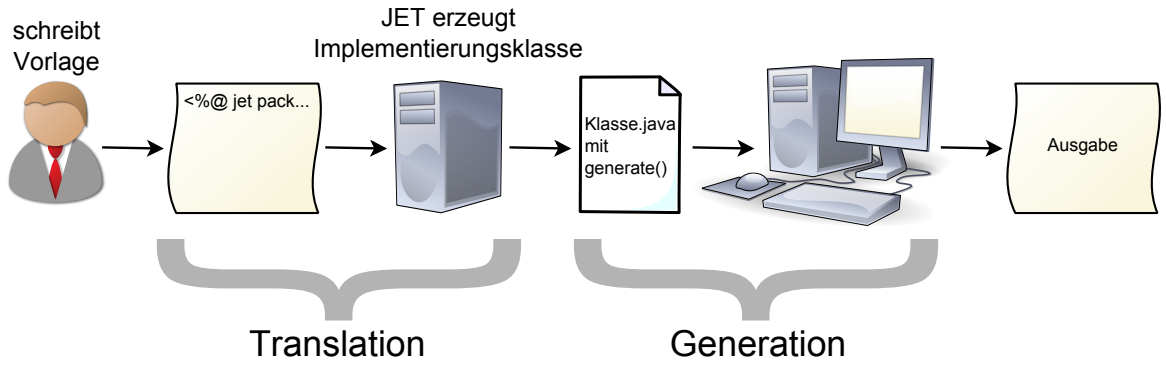

Abbildung 4.1: JET-Prozess [SS05, S. 6]

Ausdrücken (vgl. [SS05], S. 6):

- Scriptlets starten mit `<%` und enden mit `%>`. Sie können jede Art von Java-Quelltext enthalten.
- Ausdrücke (engl. expressions) erlauben das Einfügen von Text in die JET-Ausgabe.
- Direktiven (engl. directives) definieren die Einstellungen für die jeweilige JET-Vorlage.

JET verwendet zu Erstellung der Implementierungsklasse eine Art „Datei-Skelett“ (engl. skeleton). In dieser wird der Aufbau der Implementierungsklasse festgelegt. Es lassen sich z.B. die Methoden oder Methodensignaturen festlegen. Man kann auch Interfaces definieren, die die Implementierungsklasse implementieren soll. Diese kann wahlweise vom Benutzer erstellt werden. Falls vom Benutzer keine solche Datei in den Direktiven der Vorlage angegeben wird, verwendet JET eine Standard-Datei. Diese gibt nur die Methode `generate` mit dem Parameter `Object` vor (vgl. [Pop04]).

Für Vorlagen-Dateien von JET gibt es folgende Namenskonvention: „Name der Klasse“.“Dateiendung des Ausgabetyps“jet (vgl. [Pop04]).

Vorteile:

- EMF und JET sind in der Standard-Eclipse-Distribution, die im Team verwendet wird bereits vorhanden.
- Es müssen keine zusätzlichen Plug-Ins oder zusätzliche Software installiert werden.
- JET ermöglicht die Generierung beliebiger Textausgaben.

- Durch die verwendete JSP-Syntax ist JET für Java- und JSP-Erfahrene leicht erlernbar.

Nachteile:

- JET benötigt in jedem Fall EMF und zumindestens ein 'Headless"-Eclipse zur Laufzeit.
- Die Zweiteilung des JET-Prozesses bringt Schwierigkeiten bei der Integration in ein eigenes Programm oder Plug-In mit sich.

4.2.3 EMFT und JET2

JET2 ist eine neue erweiterte Version von JET. Es ist Bestandteil von Eclipse Modeling Framework Technology (EMFT). EMFT enthält neue Eclipse-Techniken, -Bibliotheken und -Plug-Ins, die sich noch in der Entwicklung befinden und die Version 1.0 noch nicht erreicht haben (vgl. [AM06]).

In JET2 wird das Konzept der Vorlagen-Technik mit JSP-Syntax durch Tag-Bibliotheken erweitert. Es gibt vorgefertigte Tag-Bibliotheken und es besteht die Möglichkeit eigene zu erstellen. Durch Verwendung dieser Bibliotheken ist es möglich den Schritt der Übersetzung in eine Implementierungsklasse zu überspringen. Man übergibt direkt einer Vorlagen-Datei eine XML-Datei als Eingabeparameter. Je nach Inhalt der XML-Datei, können beliebig viele Ausgaben in einem Vorgang generiert werden. Bei JET wäre dies nur durch wiederholte Aufrufe der Generate-Methode mit unterschiedlichen Parametern möglich (vgl. [AM06]).

Vorteile:

- Die Einsatzmöglichkeiten von JET2 werden durch die Einführung der Tag-Bibliotheken vielfältiger.
- Der Zwischenschritt der Erstellung von Implementierungsklassen entfällt.

Nachteile:

- JET2 befindet sich noch in der Entwicklung.
- Für den Einsatz von JET2 müssen zusätzliche Programmbibliotheken und Plug-Ins installiert werden.
- JET2 benötigt zur Generierung immer noch eine Eclipse-Umgebung.

4.2.4 Auswahl einer Alternative

JET ist eine bewährter Quelltextgenerator. Er wird seit vielen Jahren im EMF eingesetzt. Zur Zeit in der aktuellen Version 2.4.0. Für den Einsatz von JET muss keine zusätzliche Software installiert werden und die Einarbeitung ist durch die Verwendung bekannter Syntax in absehbarer Zeit zu schaffen. JET ermöglicht die Generierung aller Mapping-Typen. Durch die Fülle an Vorteilen wurde die Entscheidung zugunsten von JET getroffen.

4.3 Benutzeroberfläche und Einbindung in das bestehende System

Nachdem im vorigen Abschnitt die Wahl des Quelltextgenerators getroffen wurde, soll in diesem Abschnitt die grafische Oberfläche im Vordergrund stehen.

4.3.1 Art der Benutzeroberfläche

Die Entscheidung für eine Technik zur Erstellung der Benutzeroberfläche wurde durch verschiedene Voraussetzungen beeinflusst. Im Projektteam wird als Entwicklungsumgebung ausschließlich Eclipse verwendet. Sämtliche Dateien liegen dem Entwickler in Form von geöffneten Java-Projekten vor. Es bietet sich somit an, dass Mapping-Werkzeug oder zumindest die Benutzeroberfläche in Form eines Eclipse-Plug-Ins zu realisieren. Dadurch lässt sich das Mapping-Werkzeug direkt in die Entwicklungsumgebung integrieren, die jeder im Team nutzt. Man muss kein externes Programm starten, wenn Mappings generiert werden sollen. Ein weiterer Vorteil besteht darin, dass beim Entwickeln der Oberfläche auf vordefinierte Programmelemente von Eclipse oder anderen Plug-Ins zurückgegriffen werden kann. Das sind zum Beispiel Dialoge zum Laden, Speichern oder Erstellen von Dateien. Sogar Klassen für Hauptelemente von Editoren sind zum ableiten vorhanden (vgl. [GB04], S. 25 ff.).

Durch diese vielen Vorteile wurde entschieden die Komponente für die Benutzeroberfläche des Mapping-Werkzeugs als Eclipse-Plug-In zu implementieren.

4.3.2 Technik der Benutzeroberfläche

Laut der Zielsetzung der Diplomarbeit, soll das Mapping-Werkzeug dem Entwickler die Möglichkeit geben, eine Beziehung zwischen Attributen aus dem Geschäftsobjekt und der, in der Benutzeroberfläche dargestellten Komponenten herzustellen. Aus diesem Grund soll ein Editor entwickelt werden, der dies grafisch ermöglicht. Ein grafischer Editor ist übersichtlich und kann dem Benutzer die zu verbindenden Komponenten auf einen Blick darstellen.

Die Implementierung eines grafischen Editors ist eine umfangreiche Aufgabe. Man benötigt verschiedene Funktionen:

- Zum Laden und Speichern der Diagramme oder Modelle.
- Zum Auschneiden, Kopieren, Editieren und Einfügen der grafischen Elemente.
- Zum Markieren, Selektieren und Verschieben (Drag and Drop) von Elementen.

Graphical Editing Framework

Das Graphical Editing Framework (GEF) kann bei einer solchen Aufgabe helfen. Es ermöglicht die Erstellung eines grafischen Editor-Plug-Ins in kurzer Zeit. Dabei können eine Reihe von Standardoperationen von GEF genutzt werden. Dazu gehören zum Beispiel, die Bereitstellung einer Editorpalette, Funktionen zum Rückgängigmachen/Wiederherstellen von Editoroperationen und Zooming (vgl. [E05]).

Draw2D

Die gesamte grafische Darstellung des Editors wird mithilfe von Draw2D realisiert. Draw2D ist ein Framework zum Zeichnen von 2D-Standardfiguren. Es baut auf dem Standard Widget Toolkit (SWT) von Eclipse auf, ist aber im Gegensatz zu GEF unabhängig von der Eclipse-Umgebung nutzbar (vgl. [B04], S. 88).

4.4 Wiederverwendbarkeit der Mappingkonfigurationen

Wie im Abschnitt 3.2 bereits erwähnt wurde, besteht eine der Anforderungen darin die Wiederverwendbarkeit der erstellten Mappingkonfigurationen zu gewährleisten. In den folgenden Abschnitten werden Möglichkeiten zur Erreichung dieses Ziels vorgestellt. Im Anschluß wird die Auswahl einer der Möglichkeiten begründet.

4.4.1 XML

Die Speicherung einer Mappingkonfiguration in einer XML-Datei erfordert zunächst die Definition eines eigenen Dokumenttypen. Nur dadurch kann die XML-Datei auf Gültigkeit (engl. valid XML document) geprüft werden. Die Dokumenttypdefinition kann mithilfe einer Document Type Definition (DTD) oder eines XML-Schemas erfolgen. Die Definition des Dokumenttyps mit einer DTD gibt es schon seit der ersten Standardisierung von XML. Eine Definition mittels DTD kann innerhalb eines XML-Dokumentes oder durch eine externe Dokumenttypdefinition erfolgen, welche dann auf das XML-Dokument verweist (vgl. [HN05b], S. 472).

Seit 2001 gibt es als Alternative zur DTD das XML-Schema oder XML Schema Definition (XSD). Die Nutzung von XML-Schema hat mehrere Vorteile. Das XML-Schema ermöglicht eine genauere Definition von Elementinhalten. Dazu sind im XML-Schema-Standard viele primitive Datentypen definiert. Zusätzlich können zusammengesetzte Datentypen aus den Vorhandenen gebildet werden. Bei Verwendung des XML-Schema-Standards, kann für jedes Element und Attribut generell der gültige Wertebereich festgelegt werden (vgl. [HN05b], S. 472-476). Wegen der genannten Vorteile, wäre eine Nutzung des XML-Schema-Standards zu bevorzugen.

Bei der Verwendung von XML ist noch die Frage offen, wie man aus Java auf XML-Dateien zugreift. Dafür gibt es eine Fülle von möglichen Techniken. Die wichtigsten sind seit Version 1.4 in der Java 2 Standard Edition (J2SE) als Programmbibliothek Java API for XML Processing (JAXP) enthalten. Das sind die Programmierschnittstellen Document Object Model (DOM) und Simple API for XML (SAX) sowie XSL Transformation (XSLT). Eine weitere Empfehlung in diesem Bereich ist Java DOM (JDOM), eine Open-Source-Java-Bibliothek bei der, der DOM-Standard speziell an Java angepasst wurde (vgl. [HN05b], S. 479 ff.).

Bei Nutzung von XML für die Speicherung der Mappingkonfigurationen, würde sich noch eine weitere Möglichkeit an anderer Stelle eröffnen. Während der Erstellung der Mappingkonfiguration in einem grafischen Editor, könnte parallel eine Source-Ansicht zur Verfügung gestellt werden. Wenn der Benutzer die Verbindungen zwischen Geschäftsobjektattributen und Eingabemaskenelementen erstellt, würde synchron die Source-Ansicht mit dem XML-Äquivalent gefüllt werden. Dadurch hätte der Benutzer die Möglichkeit die Zuordnungen in XML zu schreiben oder grafisch mit dem Editor erstellen.

Vorteile:

- XML ist ein universell lesbarer Standard.
- Die Mappingkonfiguration im XML-Format eignet sich gut für die Weiterleitung an andere Komponenten oder Systeme.

Nachteile:

- Es muss vieles neu implementiert werden.
- Unter Umständen sind neue XML-Techniken zu installieren.

4.4.2 Serialisierung des GEF-Modells

Jeder GEF-Editor baut auf einem formalen Modell auf. Dieses Modell enthält alle Daten des Editors; d.h., inhaltliche- und auch grafische Daten, die für die Darstellung notwendig sind (vgl. [B04], S. 135 ff.). Es ist einfach diese Daten durch implementieren vom Interface `Serializable` in allen Modellklassen und der Speicherung mit einem `OutputStream` festzuhalten.

Vorteile:

- Es entsteht nur ein geringer Implementierungsaufwand.
- Es sind keine weiteren Techniken und Programmbibliotheken notwendig.

Nachteile:

- Die gespeicherten Dateien sind, im Gegensatz zu XML, außerhalb des Editors nicht verwendbar.

4.4.3 Auswahl einer Möglichkeit

Die Serialisierung des GEF-Modells erfüllt alle Anforderungen der Wiederverwendbarkeit der Mappingkonfigurationen. Da zusätzlich noch der Implementierungsaufwand gering ist, wurde diese Alternative ausgewählt.

4.5 Systementwurf

In den vorhergehenden Abschnitten dieses Kapitels wurden vor allem Techniken und Vorgehensweisen beschrieben und ausgewählt, die für die Lösung der gestellten Anforderungen optimal geeignet sind. Im folgenden wird ein Systementwurf für das zu entwickelnde Mapping-Werkzeug vorgestellt. Dabei wird anhand eines typischen Generierungsdurchlaufs, die Funktion der einzelnen Komponenten und ihr Zusammenwirken erläutert, sowie das Generierungs- und Vererbungskonzept aus Abschnitt 4.1 weiterführend geklärt.

Das System besteht aus zwei Hauptkomponenten. Der Mapping Editor stellt die Benutzeroberfläche bereit. Er wird als Eclipse-Plug-In implementiert. Die Gründe dafür wurden bereits in Abschnitt 4.3.1 erläutert. Die Komponente Mapping Generator ist zuständig für das Analysieren der Geschäftsobjekte, die Entscheidungsfindung der Attribut-Mapping-Typauswahl, die Generierung des Quelltextes und das Erstellen und Ablegen der Klassendateien.

Ein typischer Durchgang der Erstellung von Mappings läuft folgendermaßen ab. Nachdem der Benutzer den Editor gestartet hat, hat er die Möglichkeit, eine bestehende Mappingkonfiguration zu laden. Die Dateien dafür befinden sich in einem Standard-Projekt im Eclipse-Workspace. Die Dateien enthalten einen gespeicherten Zustand, des dem Editor zugrundeliegenden formalen Modells. Wenn der Benutzer keine Mappingkonfiguration laden möchte, wird ihm ein Wizard angezeigt, der durch die Erstellung einer neuen Konfigurationsdatei führt. Der Wizard wird von `org.eclipse.jface.wizard.Wizard` abgeleitet. Nach Erstellung der Datei, wählt der Benutzer das gewünschte Geschäftsobjekt und die Eingabemaske aus. Auch bei den hier benötigten Dialogen zum Laden, wird wieder auf vorhandene Dialoge von Eclipse zurückgegriffen. Das gewählte Geschäftsobjekt wird unter Zuhilfenahme der Klassen aus dem Mapping Generator analysiert. Per Reflection API werden die Zugriffsmethoden der Attribute analysiert. Die XML-Datei der Eingabemaske wird mithilfe von JDOM ausgelesen.

JDOM wurde bereits in Abschnitt 4.4.1 erwähnt. Es ist durch seine einfache Handhabung und die ideale Einbindung in Java eine gute Wahl zur Lösung dieser Problematik. Das Editor-Modell wird dann mit den gewonnenen Informationen erzeugt, von GEF in Draw2D-Objekte umgewandelt und dem Benutzer angezeigt.

Nun können Verbindungen zwischen den Geschäftsobjektattributen und den Eingabemaskenelementen erstellt werden. Dies geschieht durch einen Klick auf das Attribut. Die Maustaste wird gehalten und über dem gewünschten Maskenelement losgelassen. In Geschäftsobjekten können Attribute vorkommen, die als Dateityp einen komplexen Typen haben. In Abschnitt 5.1.1 wird im Zusammenhang mit der Klasse `ComplexTypeChecker` grundsätzliches dazu erklärt. Wenn der Benutzer einen solchen komplexen Typ verbinden will, kann er diesen „aufklappen“. In dem Fall, werden anhand des vorliegenden Full Qualified Class Names, die Informationen, der für ein Mapping geeigneten Attribute nachgeladen. Das passiert in der gleichen Weise, wie beim ursprünglichen Geschäftsobjekt. Der Benutzer kann jetzt Eingabemaskenelemente mit den Geschäftsobjektattributen des komplexen Typen verbinden. Wenn dieser wiederum selbst komplexe Typen enthält, können auch sie in derselben Art nachgeladen und verbunden werden. Dem Benutzer ist es möglich wieder auf das darüberliegende Geschäftsobjekt zu wechseln. Die im darunterliegenden komplexen Typ gemachten Verbindungen zu den Maskenelementen, werden im darüberliegenden immer noch angezeigt.

Für jedes Eingabemaskenelement hat der Benutzer die Möglichkeit einen AM-Typ auszuwählen. Falls kein Typ ausgewählt wird, wählt der Mapping Generator einen empfohlenen AM-Typ aus. Die Empfehlung wird anhand des Elementtyps festgelegt. Ist das Element zum Beispiel eine `ComboBox`, wird ein `ComboDataMapping` generiert.

Nachdem im Editor alle Zuordnungen gemacht worden, kann der Benutzer den Ausgabepfad eingeben. Hier kommt wieder ein Standard-Dialog aus der Eclipse-Programmbibliothek zum Einsatz. Dann werden die im Editor-Modell gespeicherten Zuordnungsinformationen in ein Objekt der Klasse `MappingConfig` übertragen. Zusammen mit dem Ausgabepfad bekommt der Mapping Generator die Mappingkonfiguration übergeben. Daraus generiert er dann für jede Zuordnung ein AM und für das gewählte Geschäftsobjekt und jeden untergeordneten komplexen Typ, dessen Attribute mit angezeigten Maskenelementen verbunden worden, ein OM. Quelltext 4.2 zeigt das Beispiel eines generierten AM für das Geschäftsobjekt `CltStmtRequest` und dessen Attribut `BestellerImHauseAdresse`.

```
1 public abstract class CdmCltStmtRequestBestellerImHauseAdresse extends
      ComboDataMapping<java.lang.String, CltStmtRequest>
2 {
3   public CdmCltStmtRequestBestellerImHauseAdresse(java.lang.String
        symbolicName)
4   {
5     super(symbolicName, java.lang.String.class);
6   }
7
8   @Override
9   protected java.lang.String getValue(final CltStmtRequest businessObject
        , final IMappingContext context)
10  {
11    return businessObject.getBestellerImHauseAdresse();
12  }
13
14  @Override
15  protected void setValue(final java.lang.String bestellerImHauseAdresse,
        final CltStmtRequest businessObject, final IMappingContext context)
16  {
17    businessObject.setBestellerImHauseAdresse(bestellerImHauseAdresse);
18  }
19
20  @Override
21  public abstract ITableData createTableData(final CltStmtRequest
        businessObject, final IMappingContext context);
22 }
```

Quelltext 4.2: generiertes Attribut-Mapping

Bei diesem Beispiel wurden die standard Get- und Set-Methoden generiert. Da es sich um ein `ComboDataMapping` handelt, wurde zusätzlich die Methode `createTableData` erzeugt. Weiterhin fällt auf, dass es sich um eine abstrakte Klasse handelt. Der Benutzer muss dadurch bei seiner eigenen Ableitung die Methode selbst implementieren. Alle Typen in der Klasse werden mit ihrem Full Qualified Class Name angegeben. Dadurch wird verhindert, dass bei Nutzung von zwei unterschiedlichen Klassen aus zwei verschiedenen Packages mit dem gleichen Namen ein Fehler auftreten würde.

Quelltext 4.3 zeigt ein generiertes OM. Aus Gründen der Übersichtlichkeit, wurde hier nur ein symbolischer Bezeichner verwendet und zwei AMs eingefügt. Der Bezeichner heißt auch zur Vereinfachung `SYMBOLISCHER_BEZEICHNER`.

```
1 public abstract class BusinessObjectGenTestMapping extends
    AbstractObjectMapping
2 {
3   public final String SYMBOLISCHER_BEZEICHNER = "SYMBOLISCHER_BEZEICHNER"
      ;
4
5   protected BusinessObjectGenTestMapping()
6   {
7     // abstract Mapping
8     addMapping(getCdmCltStmtRequestBestellerImHauseAdresse());
9     // konkretes Mapping
10    addMapping(getSmCltStmtRequestBestellerPerId());
11  }
12
13  /**
14   * SYMBOLISCHER_BEZEICHNER
15   */
16  protected abstract CdmCltStmtRequestBestellerImHauseAdresse
      getCdmCltStmtRequestBestellerImHauseAdresse();
17
18  /**
19   * SYMBOLISCHER_BEZEICHNER
20   */
21  protected SmCltStmtRequestBestellerPerId
      getSmCltStmtRequestBestellerPerId()
22  {
23    return new SmCltStmtRequestBestellerPerId(SYMBOLISCHER_BEZEICHNER);
24  }
25 }
```

Quelltext 4.3: generiertes Objekt-Mapping

Zu Beginn der Klasse werden die symbolischen Bezeichner als String-Konstanten definiert. Danach folgt der Konstruktor, der mithilfe der Methode `addMapping` die AMs hinzufügt. Die AMs werden nicht direkt als Parameter in `addMapping` erzeugt, sie werden als Methode entkoppelt. Nur so können auch die abstrakten AMs im generierten OM, bereits eingebunden werden. Die entkoppelten Methoden zur Erzeugung der AMs folgen im Anschluss an den Konstruktor. Bei jeder Methode wird im Kommentar der zugeordnete symbolische Bezeichner angegeben. Nur so kann der Benutzer bei seiner eigenen Ableitung noch wissen, welcher symbolische Bezeichner zu welchem Mapping gehört.

Man hat jetzt also die generierten AMs, wie in Quelltext 4.2 gezeigt und ein oder

mehrere OMs, wie in 4.3. Um nun die Mappings im System einsetzen zu können muss man nur noch das OM ableiten.

In Quelltext 4.4 wird das Beispiel aus Quelltext 4.3 weitergeführt, indem wie erwähnt, das generierte OM abgeleitet wird.

```
1  public class BusinessObjectTestMapping extends
       BusinessObjectGenTestMapping
2  {
3    private static final BusinessObjectTestMapping INSTANCE = new
       BusinessObjectTestMapping();
4
5    public static BusinessObjectTestMapping getInstance()
6    {
7      return INSTANCE;
8    }
9
10   @Override
11   protected CdmCltStmtRequestBestellerImHauseAdresse
       getCdmCltStmtRequestBestellerImHauseAdresse()
12   {
13     return new CdmCltStmtRequestBestellerImHauseAdresse(
         SYMBOLISCHER_BEZEICHNER)
14     {
15       @Override
16       public ITableData createTableData(CltStmtRequest businessObject,
           IMappingContext context)
17       {
18         String[] s = {"tabellen", "inhalt"};
19         return new TableData(s);
20       }
21     };
22   }
23
24   public Class<?> getBusinessObjectClass()
25   {
26     return BusinessObjectTestMapping.class;
27   }
28 }
```

Quelltext 4.4: abgeleitetes Objekt-Mapping

Zunächst wird das OM als Singleton-Pattern implementiert, um sicherzustellen das zur Laufzeit nur eine Instanz des Objekts erzeugt wird. Dazu definiert man ein Static-

Attribut `INSTANCE`. So muss das Objekt beim erstmaligen Zugriff auf die Klasse, durch die JVM instanziert werden. Durch die Methode `getInstance`, kann auch von außerhalb der Klasse eine Instanz angefordert werden. Als nächstes folgt eine mögliche Implementierung der abstrakten Methode aus der Oberklasse, die hier ausprogrammiert wird. Im Beispiel wird eine extrem vereinfachte Form gezeigt, die nur ein Objekt vom Typ `ITableData` zurückliefert. Zum Schluss muss nur noch eine Methode `getBusinessObjectClass` geschrieben werden, die das Class-Objekt der OM-Klasse zurückgibt.

Kapitel 5

Implementierung

5.1 Mapping Generator

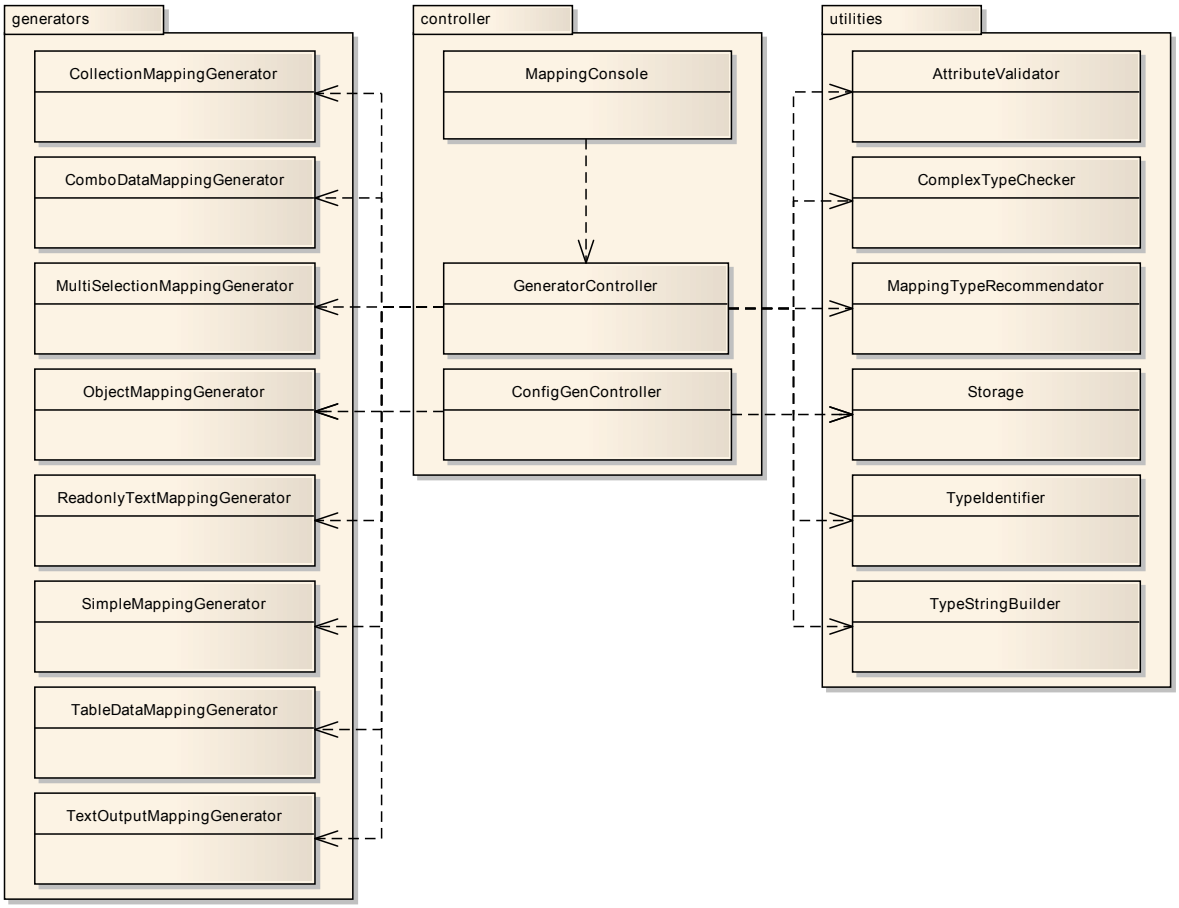

Abbildung 5.1: Klassendiagramm Mapping Generator

5.1.1 Das Package Utilities

Das Package `Utilities` enthält, wie der Name bereits sagt, Utility- und Hilfsklassen zum Laden, Analysieren und Verarbeiten der Informationen aus Geschäftsobjekten und Eingabemasken. Zudem stehen Funktionen zum Speichern von generierten Quelltexten als Java-Klassen zur Verfügung.

Die Klasse AttributeValidator

Die Klasse `AttributeValidator` enthält Methoden zum Analysieren von Geschäftsobjekten. Ziel ist es, die verwendbaren Attribute der Klasse zu finden, die sich für das Mapping eignen. Zu diesem Zweck werden per Reflection API die Methodeninformationen ausgelesen. Das hat mehrere Gründe. Zum einen handelt es sich bei den Geschäftsobjekten um Java-Beans; d.h., es gibt immer Zugriffsmethoden für die jeweiligen Attribute. Zum anderen lassen sich anhand der Zugriffsmethoden weitere Informationen gewinnen. Hat ein Attribut beispielsweise keine Set-Methode ist es automatisch „Readonly“. Beim späteren Erstellen der Mappings wird diese Information u.a. benötigt, um die richtige AM-Klasse zu generieren.

Des Weiteren hilft die Analyse der Methodeninformationen, die richtigen Attribute des Geschäftsobjekts auszusortieren. Zu diesem Zweck wurden erweiterte Bean-Konventionen für die Methoden aufgestellt:

- Die Methode muss **`public`** sein.
- Der Methodenname beginnt mit „get“, es gibt keinen Parameter und einen Rückgabewert.
- Der Methodenname beginnt mit „is“, es gibt keinen Parameter und einen Rückgabewert.
- Der Methodenname beginnt mit „set“, es gibt einen Parameter und keinen Rückgabewert. Zu dieser Set-Methode muss eine zugehörige Get- oder Is-Methode existieren.

`GetValidatedMethods` liefert eine Liste mit `Method`-Objekten zurück, die nach den besagten Kriterien ausgewählt wurden. Das Weitergeben der Informationen in Form von Objekten der Reflection-Klasse `Method` hat den Vorteil, beim späteren Weiterverarbeiten immer noch auf alle zusätzlichen Informationen der gefundenen Methode zurückgreifen zu können.

Quelltext 5.1 zeigt ausführlich kommentiert die Methode `getBeanMethods`, die für das finden der Geschäftsobjektmethoden nach den aufgestellten Kriterien verantwortlich ist.

```
1  private ArrayList<Method> getBeanMethods(Method[] methods, boolean setter
       )
2  {
3    ArrayList<Method> valid = new ArrayList<Method>();
4    for(Method method : methods)
5    {
6      if(method.getName().startsWith("set") && setter)
7      {
8        // hat keinen RueckgabeWert und einen Parameter
9        if((method.getGenericReturnType().toString().equals("void"))
10           && (method.getGenericParameterTypes().length == 1)
11       )
12       {
13         // es gibt eine zugehoerige Get- oder Is-Methode
14         if(foundBelongingAccessMethod(method))
15         {
16           valid.add(method);
17         }
18       }
19     }
20
21     if((method.getName().startsWith("get") || method.getName().startsWith
           ("is"))
22         && setter == false)
23     {
24       // hat keinen Parameter und einen RückgabeWert
25       if((method.getGenericParameterTypes().length == 0)
26           && (method.getGenericReturnType().toString() != "void")
27       )
28       {
29         if(method.getName() != "getClass")
30         {
31           if(method.getName().startsWith("is"))
32           {
33             /*
34              * Prüfung ob die Bean-Konvention bezüglich is eingehalten
                    wird.
35              * Is darf nur verwendet werden wenn der Rückgabetyp der
36              * primitve Datentyp boolean ist.
37              */
```

```
38              Type type = method.getGenericReturnType();
39              String typeString = TypeIdentifier.idType(type);
40
41              if(typeString.equals("SimpleType"))
42              {
43                Class<?> typeClass = (Class<?>) type;
44                if(typeClass.isPrimitive() && typeClass.getName().equals("
                      boolean"))
45                {
46                  valid.add(method);
47                }
48              }
49            }
50            else
51            {
52              valid.add(method);
53            }
54          }
55        }
56      }
57    }
58    return valid;
59  }
```

Quelltext 5.1: AttributeValidator Methode getValidatedMethods

Die Klasse MappingTypeRecommendator

Die Klasse `MappingTypeRecommendator` stellt Methoden zum Finden der möglichen AM-Typen für die Generierung zur Verfügung. Nachdem die Klasse `AttributeValidator`, wie im vorigen Abschnitt beschrieben, die Methodeninformationen zu den Attributen eines Geschäftsobjekts herausgefunden hat, kann nun die Methodeninformation an die Klasse `MappingTypeRecommendator` weitergegeben werden. Anhand des Rückgabetyps der Get-Methoden, der dem Dateityp des Attributes entspricht, wird eine AM-Typ-Empfehlung in Form einer `ArrayList<String>` zurückgegeben.

Die Klasse ComplexTypeChecker

Die Klasse `ComplexTypeChecker` ist für die Erkennung von komplexen Typen zuständig. Komplexe Typen sind solche, die ein eigenes OM haben bzw. bekommen sollen. Geschäftsobjekte können also Attribute haben, die komplexe Typen als Dateitypen haben. Zum Beispiel hat das Geschäftsobjekt `Geschäftsbeziehung` ein Attribut `vertragsGegenpartner`, dessen Datentyp `Partner` ist und für `Partner` existiert wiederum ein eigenes OM mit Namen `PartnerMapping`. Zur Erkennung dieser Klassen arbeitet der ComplexTypeChecker nach dem Ausschlussverfahren. Primitive Datentypen und Typen wie `String`, `BigInteger` und andere sind in einer Liste aufgeführt. Alle Typen, die nicht mit den Listentypen übereinstimmen, werden als komplexe Typen erkannt.

Die Klasse Storage

Die Klasse `Storage` enthält eine Methode zum Speichern der Generator-Ausgaben in Java-Klassen. Zuvor wird die Verzeichnisstruktur des Geschäftsobjekt-Packages nachgebildet zudem das Mapping gehört. Dadurch kann beim späteren weiterverwenden, in einer Jar-Datei, die mit den generierten Klassen gefüllt wird, die Package-Struktur für den Classpath eingehalten werden. Die eigentliche Speicherung erfolgt mit einem `BufferedWriter` inklusive `FileWriter`. Die genaue Funktionsweise kann in Quelltext A.1 eingesehen werden.

Die Klasse TypeIdentifier

Die Klasse `TypeIdentifier` ist eine kleine Utility-Klasse zum genaueren Identifizieren eines `java.lang.reflect.Type`. Diesen bekommt man bei der Arbeit mit der Reflection API oft als Rückgabewert. Da jedoch vor der weiteren Bearbeitung, zunächst erst einmal der genauere Typ bestimmt werden muss, wird der `TypeIdentifier` häufig verwendet.

Exkurs - Type-Interface-Problematik

Ein Beispiel für die Arbeit mit `java.lang.reflect.Type`: Man hat ein `Method`-Objekt. Dieses enthält Informationen über eine Get-Methode eines Attributes von einem Geschäftsobjekt. Wenn man den Rückgabetyp dieser Methode erhalten will, ruft man

`getGenericReturnType()` auf und bekommt einen `java.lang.reflect.Type`. Bei diesem handelt es sich um ein Interface, das keine Methoden implementiert. Eigentlich wird eine `TypeImpl` zurückgegeben. Diese stellt nur die Methoden von `Object` bereit. Allein mit der Methode `toString`, bekommt man weitere brauchbare Informationen. Nämlich den Full Qualified Class Name des obersten Typs, der sich in der `TypeImpl` befindet. Wenn es sich zum Beispiel um einen Rückgabetyp `Collection<String>` handelt, bekommt man nur die Ausgabe `java.util.Collection`. Der Datentyp, den die Collection enthalten darf, bleibt verborgen.

Die Methode `idType`, der Klasse `TypeIdentifier`, liefert genauere Informationen des obersten Typs. Das erlaubt die ungefähre Kategorisierung und Weiterverarbeitung des Typs. Im Fall der vorher erwähnten Collection, würde ein `ParameterizedType` gemeldet. Durch diese Information, kann jetzt der einfache `java.lang.reflect.Type` in einen `ParameterizedType` durch „Cast“ umgewandelt werden. Nun ist man in der Lage, durch die Methoden des `ParameterizedType`, diesen weiter zu untersuchen. Zum Beispiel durch die Methode `getActualTypeArguments`, die den inneren Typ der Collection liefern würde. Dieser wird wieder als `java.lang.reflect.Type` zurückgegeben. Im Fall einer stark verschachtelten Collection oder Map, müsste man nun wieder beginnen, diesen auf die gleiche Art zu untersuchen.

Lösung TypeStringBuilder

Für eine solche Situation, bietet sich in der Programmierung eine Lösung in Form einer Rekursion an. Diese Lösung wurde in der Klasse `TypeStringerBuilder` implementiert. In Quelltext A.2 nachzulesen.

5.1.2 Das Package Controller

Das Package `Controller` enthält Klassen zur Steuerung des Generierungsvorgangs.

Die Klasse GeneratorController und MappingConsole

Zu Beginn der Implementierung wurde zunächst die Komponente Mapping Generator fertiggestellt. So konnten die Generierungstechniken getestet und zu einem gegebenen Geschäftsobjekt alle möglichen AM's erstellt werden, ohne schon Eingabemasken bzw. deren Elemente angeben zu müssen. Der Mapping Generator kann als eigenständige

Konsolenanwendung genutzt werden. Für die Bedienung per Kommandozeile wurde die Klasse `MappingConsole` erstellt. Sie nimmt die Eingabeparameter, also den Full Qualified Class Name und den Ausgabepfad entgegen. Danach wird ein Generatordurchgang mit dem `GeneratorController` angestossen.

Der `GeneratorController` steuert den kompletten Durchgang der Mappinggenerierung. Er greift dabei auf die Klassen des Packages `Utilities` und auf die Generatorklassen des Packages `Generators` zu.

Die Klasse ConfigGenController

Nach der Fertigstellung der Mapping-Editor-Komponente, musste ein „Verbindungselement“ zwischen den beiden Komponenten erstellt werden. Denn der Mapping Generator sollte auch die Generierung der im Editor erstellten Konfigurationen übernehmen. Zu diesem Zweck wurde die Klasse `ConfigGenController` entwickelt. Sie nimmt die erstellte Konfiguration in Form eines Objekts der Klasse `MappingConfig` und den gewünschten Ausgabepfad entgegen. Der Generierungsvorgang findet wieder mithilfe der anderen Packages des Mapping Generator statt.

5.1.3 Das Package Generators

Das Package `Generators` enthält die Generatorklassen, die von JET aus den Vorlagen erstellt wurden. Zum OM und zu jedem AM-Typ gibt es jeweils eine Generatorklasse. Im nächsten Abschnitt folgen weitere Informationen zu diesem Thema.

5.2 Mapping Templates

Die Komponente Mapping Templates wird benötigt, falls die Generator-Klassen des Mapping Generators verändert werden sollen.

5.2.1 Templates

Hier sind alle Vorlagen für die Generierung der Implementierungsklassen enthalten. Bei Veränderung der Vorlagen, werden beim Speichern die Java-Klassen im Package `src-gen` automatisch neu generiert und von der sogenannten „JET-Engine“ überschrieben.

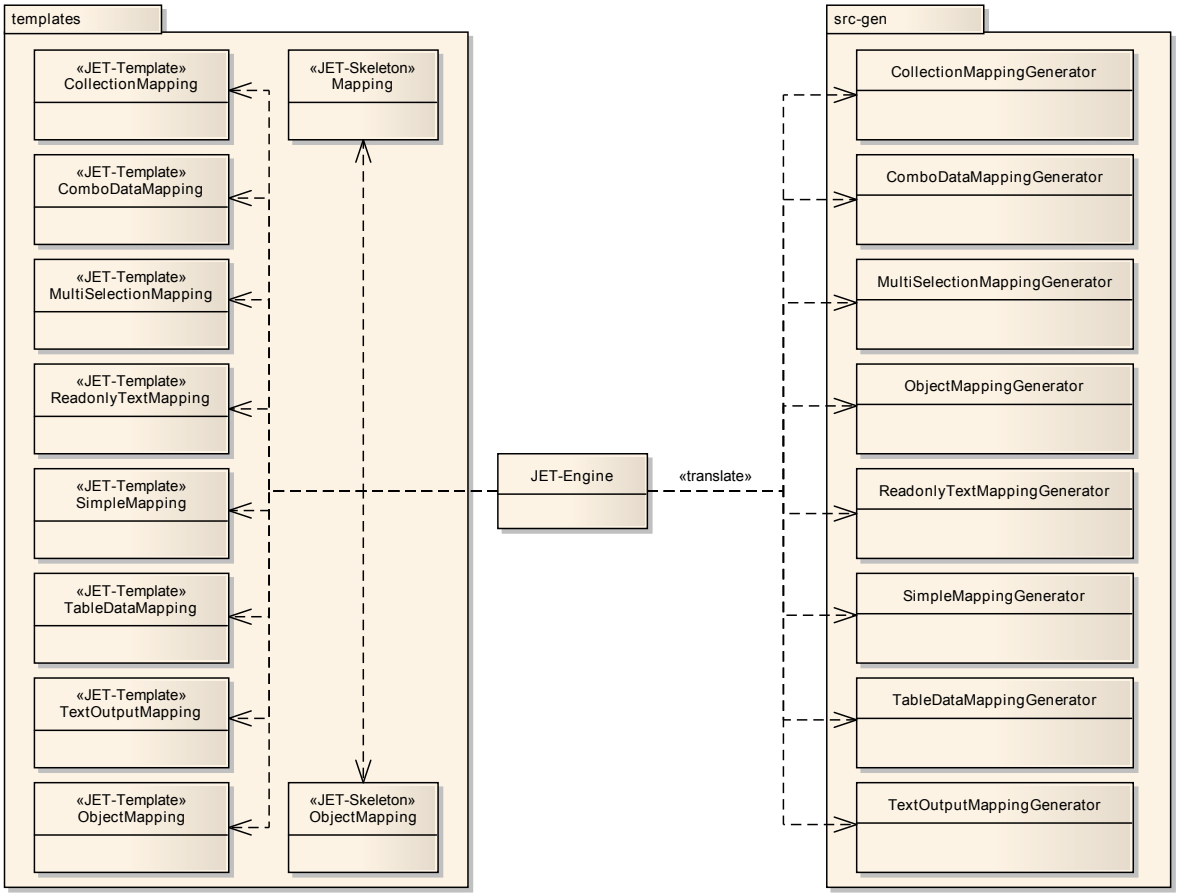

Abbildung 5.2: Mapping-Templates-Übersicht

Beim Programmieren der Vorlagen wurde darauf geachtet, möglichst wenig Programmlogik zu implementieren. Das hat den Vorteil, dass die Vorlagen seltener geändert werden müssen.

In Quelltext A.3 kann man die Vorlage für die Generatorklasse des AM-Typs Simple Mapping sehen. Es ist zu erkennen, dass die variablen Anteile (JSP-Syntax Befehle in `<%` und `%>`) gering gehalten wurden.

Um die Methodensignatur der Generate-Methode in der Generatorklasse zu definieren, werden die sogenannten „Skeletons“ verwendet. Die Vorlagen der AM-Typen-Generatoren verwenden alle das `Mapping.skeleton`. Dessen Inhalt ist im Quelltext A.4 zu finden. Die Vorlage des OM-Generators besitzt ein eigenes „Skeleton“, da die Übergabeparameter der Generate-Methode sich stark unterscheiden.

5.2.2 Das Package Src-gen

In diesem Package befinden sich die Generatorklassen, die von der „JET-Engine“ unter Verwendung der Vorlagen und „Skeleton“ generiert werden. Die Klassen werden immer nach einem gleichen Muster generiert. Die statischen Bestandteile werden zunächst als String-Konstanten implementiert. In der Generate-Methode werden dann die variablen und statischen Bestandteile mithilfe eines `StringBuffer` in der richtigen Reihenfolge aneinandergefügt. Das Ergebnis wird als `String` zurückgegeben. In Quelltext A.5 kann man das anhand der Generatorklasse des Simple Mapping nachvollziehen. Für den Entwickler ist die Implementierung dieser Klassen nicht wichtig, da man ohnehin nur die Generate-Methode für den Zweck der Quelltextgenerierung benutzt.

5.3 Mapping Editor

5.3.1 Das Package Editor

Dieses Package enthält eine Reihe von allgemeinen Klassen, die den Editor und die Plug-In-Struktur betreffen. Die Klasse `MappingEditorPlugin` stellt durch Singleton-Implementierung sicher, dass nur eine Instanz des Plug-Ins zur Laufzeit existiert. Die Klasse `MappingEditor` implementiert den eigentlichen Editor und erbt von `GraphicalEditorWithFlyoutPalette` aus der GEF-Bibliothek. Als Attribute enthält sie das zu bearbeitende `MappingDiagram` und die Editorpalette. Die Edit Part Factories (Controller-Bestandteile) werden für den Editor definiert. Die Art des „Leinwand“-Objekts auf dem später alle Objekte dargestellt werden, wird bestimmt. Hier wird der Typ `ScalableFreeformRootEditPart` verwendet. Der Benutzer kann bei diesem alle selbst gesetzten Objekte markieren, skalieren und frei verschieben. Dies wird nur durch die Hierarchie, Art und Anordnung der Objekte eingeschränkt. Des Weiteren sind im `MappingEditor` Methoden zum Laden, Speichern und Zwischenspeichern des Editor-Inhalts und der erstellten Mapping-Konfiguration realisiert.

Der Editor benötigt während der Bearbeitung eines Diagramms bzw. einer Mapping-Konfiguration, eine Datei in der der aktuelle Stand gespeichert werden kann. Dadurch kann beim nächsten Start des Editors der zuletzt bearbeitete Inhalt wieder geladen bzw. hergestellt werden. Um dies zu gewährleisten, wird nach Start des Editors und bei nicht vorhandensein einer zuletzt gespeicherten Konfigurationsdatei, zunächst der

Wizard (Klasse `FileCreationWizard`) zur Erstellung einer solchen geöffnet. Der Wizard leitet sich von der abstrakten Klasse `org.eclipse.jface.wizard.Wizard` ab und implementiert das Interface `org.eclipse.ui.INewWizard`.

5.3.2 Das Package Editor.model

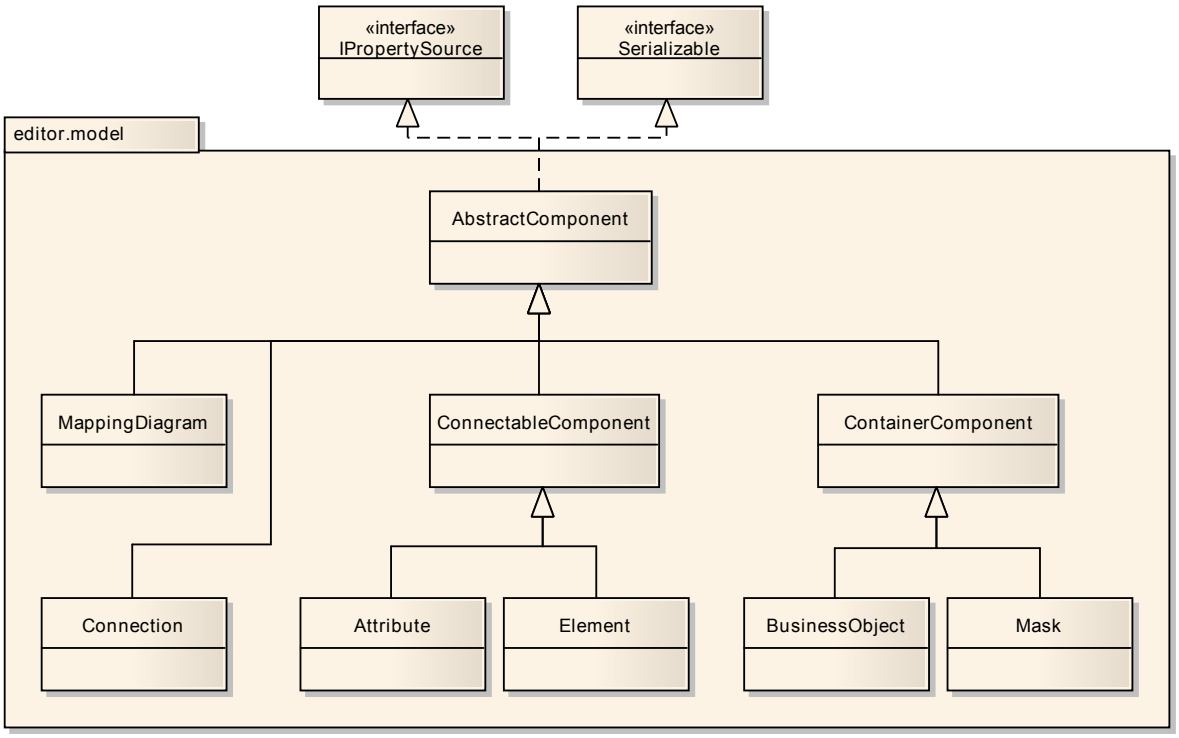

Abbildung 5.3: Mapping Editor Package editor.model

In diesem Package wird das dem Editor zugrunde liegende Modell definiert. GEF schreibt keine genauen Regeln für das Modell eines Editors vor. Es sollten nur einige Empfehlungen erfüllt werden. Das Modell sollte alle Daten und Benutzereingaben enthalten. Auch Daten, die die grafische Darstellung im Diagramm betreffen, sollen im Modell enthalten sein. Es darf keine Referenzen auf die Benutzeroberfläche oder andere Teile des Editors halten. Das wird erreicht, indem man den `PropertyChangeSupport` nutzt. Es wird eine Möglichkeit für den Controller (EditParts) geschaffen die Zustandsänderungen des Modells zu registrieren und darauf zu reagieren. Hierzu werden grundlegende Methoden in der Klasse `AbstractComponent` implementiert. Die Methode `addPropertyChangeListener`, um einen `PropertyChangeListener` zu registrieren. `RemovePropertyChangeListener`, um einen solchen wieder abzumelden und `firePropertyChange`, um eine Änderung an den Listenern mitzuteilen. Da alle anderen

Modellklassen von `AbstractComponent` erben, bekommen sie zugleich die notwendigen Methoden mitvererbt.

Die Klasse `ConnectableComponent` legt die grundlegenden Methoden und Attribute für die Modellobjekte, die miteinander verbunden werden können, fest. `Attribute` und `Element` erben von `ConnectableComponent`. Sie können im Editor als einzige Objekte miteinander verbunden werden. Dabei repräsentiert `Attribute` das Attribut eines Geschäftsobjekts und `Element` das Eingabeelement einer Eingabemaske. Ein genauerer Blick auf die wichtigen Methoden `addConnection`, `removeConnection` und andere in diesem Zusammenhang wird in Quelltext 5.2 gegeben.

```
1  public abstract class ConnectableComponent extends AbstractComponent
2  {
3    ...
4    public static final String SOURCE_CONNECTIONS_PROP = "Connectable.
         SourceConn";
5    public static final String TARGET_CONNECTIONS_PROP = "Connectable.
         TargetConn";
6    ...
7
8    private List<Connection> sourceConnections = new ArrayList<Connection
         >();
9    private List<Connection> targetConnections = new ArrayList<Connection
         >();
10
11   /**
12    * Eine Eingehende oder Ausgehende Verbindung zu dieser Komponente
          hinzufuegen.
13    *
14    * @param conn Connection not null
15    * @throws IllegalArgumentException wenn die Verbindung null ist oder
          Start und Endpunkt gleich sind.
16    */
17   void addConnection(Connection conn)
18   {
19     if (conn == null || conn.getSource() == conn.getTarget())
20     {
21       throw new IllegalArgumentException();
22     }
23
24     if (conn.getSource() == this)
25     {
26       sourceConnections.add(conn);
```

```
27        firePropertyChange(SOURCE_CONNECTIONS_PROP, null, conn);
28      }
29      else if (conn.getTarget() == this)
30      {
31        targetConnections.add(conn);
32        firePropertyChange(TARGET_CONNECTIONS_PROP, null, conn);
33      }
34    }
35    ...
36
37    public List<Connection> getSourceConnections()
38    {
39      return new ArrayList<Connection>(sourceConnections);
40    }
41
42    public List<Connection> getTargetConnections()
43    {
44      return new ArrayList<Connection>(targetConnections);
45    }
46
47    void removeConnection(Connection conn)
48    {
49      if (conn == null)
50      {
51        throw new IllegalArgumentException();
52      }
53      if (conn.getSource() == this)
54      {
55        sourceConnections.remove(conn);
56        firePropertyChange(SOURCE_CONNECTIONS_PROP, null, conn);
57      }
58      else if (conn.getTarget() == this)
59      {
60        targetConnections.remove(conn);
61        firePropertyChange(TARGET_CONNECTIONS_PROP, null, conn);
62      }
63    }
64    ...
65  }
```

Quelltext 5.2: Ausschnitte Klasse ConnectableComponent

Die Klasse `ContainerComponent` definiert auf der anderen Seite die grundlegenden Methoden, für die von ihr erbenden Klassen `BusinessObject` und `Mask`. Diese dienen als

Container für Objekte der Klasse `Attribute` und `Element`. Die Klasse `Connection` implementiert die eigentliche Verbindung und ihr aussehen als einfache, durchgezogene Linie. `MappingDiagram` ist der Gesamtcontainer für alle Modellelemente eines Diagramms bzw. einer Mapping-Konfiguration.

5.3.3 Das Package Editor.model.commands

Die „Commands“ sind die Objekte, mit denen man im eigentlichen Sinne sein Modell verändert. `ConnectionCreateCommand` kommt zum Beispiel beim Erstellen einer neuen Verbindung zum Einsatz. `ConnectableComponentEditPart` erstellt ein solches „Command“. Durch die Methode `canExecute` wird geprüft, ob die Verbindung hergestellt werden kann. Wenn das Startobjekt nicht mit dem Zielobjekt übereinstimmt und noch keine gleiche Verbindung mit den beiden Objekten besteht, wird `execute` aufgerufen und ein Objekt der Klasse `Connection` aus `editor.model` erzeugt. Die Methoden `undo` und `redo` sind bei der Gewährleistung der Rückgängig- und Wiederholen-Funktion des Editors beteiligt.

5.3.4 Das Package Editor.figures

Die Klassen in diesem Package stellen die Implementierung der Ansicht dar. Alle erben von der Klasse `Figure`, denn jedes sichtbare Element in einem Draw2D-Fenster, wird auf einem `Figure`-Objekt gezeichnet. Eine Ausnahme in diesem Package bildet noch die Klasse `PolylineConnectionFactory`. Sie liefert u.a. `PolylineConnection`-Objekte, die in der Ansicht die Verbindungen zwischen Geschäftsobjektattributen und Eingabemaskenelementen darstellen. Eine der Oberklassen von `PolylineConnection` ist ebenfalls `Figure`.

5.3.5 Das Package Editor.parts

Die `EditParts` stellen die Steuerungskomonenten des Editors dar. Sie haben u.a. die Aufgabe der Erzeugung des `Figure`-Objekts und dessen Aktualisierung bei Modelländerungen. Die `EditParts` werden mithilfe von Factories (`ComponentEditPartFactory` und `ComponentTreeEditPartFactory`) erstellt.

5.3.6 Das Package Editor.policies

Dieses Package enthält die `EditPolicies` für die `EditParts` des Editors. `EditPolicies` definieren was mit einem `EditPart` gemacht werden kann. Die Klasse `ConnectableComponentEditPolicy` legt zum Beispiel fest, was mit den für die Ableitungen von `ConnectableComponent` zuständigen `EditParts` getan werden darf. `ContainerComponentEditPolicy` definiert das gleiche für die `ContainerComponents`.

5.3.7 Das Package Config

Die Klasse `MappingConfig` ist für die Weitergabe einer in dem Editor erstellten Mapping-Konfiguration zuständig. Sie enthält als Attribut eine `Map`, die die Zuordnung von Geschäftsobjektattributen und Eingabemaskenelemente übernimmt. Die Daten sind jeweils in Objekten von `ConfigAttribute` und `ConfigElement` enthalten. Diese Klassen beinhalten alle nötigen Informationen zum Generieren der Mapping-Konfigurationen.

Kapitel 6

Zusammenfassung

In der vorliegenden Arbeit wurde die automatisierte Generierung von Mappings zwischen Geschäftsobjekten, Präsentationsschicht und Geschäftslogik betrachtet. Zu Beginn wurden die grundlegenden Begriffe und Techniken, zum Verständnis der Arbeit erklärt. Anschließend wurde die Ausgangslage der Mappings im Projekt betrachtet. Es wurden Anforderungen an das zu entwickelnde Mapping-Werkzeug formuliert. Im Kapitel Entwurf, wurden Techniken und Lösungsmöglichkeiten gegenübergestellt und daraus ein Lösungskonzept erstellt. Abschließend wurden Aspekte der Implementierung vorgestellt und erläutert.

Die Zielsetzung, ein Mapping-Werkzeug zu erstellen, mit dem der Entwickler eine Beziehung zwischen Attributen des Geschäftsobjekts und Benutzeroberflächenkomponenten herstellen kann, wurde erreicht. Auch die im Analysekapitel aufgestellten Anforderungen konnten durch den Prototyp, der im Rahmen dieser Arbeit entstanden ist, erfüllt werden.

Anhang A

Quelltextauszüge

```
1  public static void createClassFile(String javaCode, String path, String
       fileName)
2  {
3      System.out.println("Erstellt: "+path+File.separator+fileName);
4
5      try
6      {
7        // File zur Erstellung der Verzeichnisstruktur
8        File genFile = new File(path);
9
10       try
11       {
12         // Verzeichnisstruktur erstellen
13         if(genFile.mkdirs())
14         {
15           System.out.println("Verzeichnisstruktur wurde vollständig
                 erstellt.");
16         }
17       }
18       catch(SecurityException se)
19       {
20         System.err.println("Fehler beim Erstellen der Verzeichnisstruktur
               .");
21         se.printStackTrace();
22       }
23
24       BufferedWriter bufferedWriter = new BufferedWriter(
25           new FileWriter(path+File.separator+fileName));
26       bufferedWriter.write(javaCode);
```

```
27        bufferedWriter.close();
28      }
29      catch(IOException e)
30      {
31        System.err.println("Fehler beim Erstellen und Schreiben der Datei "
              +fileName+
32            " in "+path+".");
33        e.printStackTrace();
34      }
35  }
```

Quelltext A.1: Methode createClassFile aus Storage

```
1  public class TypeStringBuilder
2  {
3    private String typeString;
4    // Anzahl der Durchläufe der Rekursion
5    private int times;
6
7    public TypeStringBuilder()
8    {
9      typeString = "";
10     times = 0;
11   }
12
13   public String buildTypeString(Type t)
14   {
15     typeString = "";
16     print(t);
17     return typeString;
18   }
19
20   private void print(Type t)
21   {
22     if (t instanceof TypeVariable)
23     {
24       print((TypeVariable<?>)t);
25     }
26     else if (t instanceof WildcardType)
27     {
28       print((WildcardType)t);
29     }
30     else if (t instanceof ParameterizedType)
31     {
```

```
32        print((ParameterizedType)t);
33      }
34      else if (t instanceof GenericArrayType)
35      {
36        print((GenericArrayType)t);
37      }
38      else
39      {
40        // ersetzen des Platzhalters
41        typeString = typeString.replaceFirst("!!!",
42            ((Class<?>) t).getSimpleName());
43
44        // beim Ersten Durchlauf, einfacher Typ
45        if(times == 0)
46        {
47          typeString = ((Class<?>) t).getSimpleName();
48        }
49      }
50    }
51
52    private void print(TypeVariable<?> v)
53    {
54      for (Type t : v.getBounds())
55      {
56        print(t);
57      }
58    }
59
60    private void print(WildcardType wt)
61    {
62      // wenn LowerBounds leer ist handelt es sich um einen Extends-
          Wildcard-Ausdruck
63      if(wt.getLowerBounds().length == 0)
64      {
65        typeString = typeString.replaceFirst("!!!", "? extends !!!");
66      }
67      else
68      {
69        typeString = typeString.replaceFirst("!!!", "? super !!!");
70      }
71
72      // durch die LowerBounds laufen
73      for (Type b : wt.getLowerBounds())
74      {
```

```
75        print(b);
76      }
77
78      for (Type b : wt.getUpperBounds())
79      {
80        print(b);
81      }
82
83      // durch die UpperBounds laufen
84      for (int i = 0; i < wt.getUpperBounds().length; i++)
85      {
86        if((i + 1) == wt.getUpperBounds().length)
87        {
88          print(wt.getUpperBounds()[i]);
89        }
90      }
91    }
92
93    private void print(ParameterizedType pt)
94    {
95      if(times == 0)
96      {
97        typeString += ((Class<?>) pt.getRawType()).getSimpleName() + "<!!!>
              ";
98      }
99
100     times++;
101
102     for (int i = 0; i < pt.getActualTypeArguments().length; i++)
103     {
104       // bis zum vorletzten Durchlauf vor Ende der Iteration
105       if(typeString.contains("!!!") && (i + 1) != pt.
              getActualTypeArguments().length)
106       {
107         typeString = typeString.replaceFirst("!!!", "!!!, !!!");
108       }
109
110       if(pt.getActualTypeArguments()[i] instanceof ParameterizedTypeImpl)
111       {
112         typeString = typeString.replaceFirst("!!!",
113             pt.getActualTypeArguments()[i].toString() + "<!!!>");
114       }
115
116       print(pt.getActualTypeArguments()[i]);
```

```
117     }
118   }
119
120
121   private void print(GenericArrayType gat)
122   {
123     print(gat.getGenericComponentType());
124   }
125 }
```

Quelltext A.2: Klasse TypeStringBuilder

```
1 <%@ jet package="generators" imports="java.lang.reflect.Method" class="
    SimpleMappingGenerator" skeleton="Mapping.skeleton" %>
2 package <%=packageName%>;
3
4 /**
5  * Diese Klasse wurde automatisch generiert.
6  * Bitte den generierten Code nicht editieren.
7  * Bitte nur von dieser Klasse ableiten.
8  *
9  * Attribut-Mapping fuer das Business Object: <%=businessObjectClass.
     getSimpleName()%>
10 * Attribut: <%=attributeName%>
11 * Attribut-Mapping-Typ: SimpleMapping
12 */
13 public class Sm<%=genClassName%> extends de.innovations.objectmapping.
    Mapping<<%=returnType%>, <%=businessObjectClass.getName()%>>
14 {
15   public Sm<%=genClassName%>(java.lang.String symbolicName)
16   {
17     super(symbolicName, <%=returnType%>.class);
18   }
19
20   @Override
21   protected <%=returnType%> getValue(final <%=businessObjectClass.getName
       ()%> businessObject, final de.innovations.objectmapping.
       IMappingContext context)
22   {
23     return businessObject.<%=returnType.contains("Boolean") ? "is" : "get
         "%><%=attributeName%>();
24   }
25   <% if(readOnly == false){ %>
26   @Override
```

```
27   protected void setValue(final <%=returnType%> <%=attributeName.replace(
         attributeName.substring(0, 1), attributeName.substring(0, 1).
         toLowerCase())%>, final <%=businessObjectClass.getName()%>
         businessObject, final de.innovations.objectmapping.IMappingContext
         context)
28   {
29     businessObject.set<%=attributeName%>(<%=attributeName.replace(
           attributeName.substring(0, 1), attributeName.substring(0, 1).
           toLowerCase())%>);
30   }<% } %>
31 }
```

Quelltext A.3: Template Simple Mapping

```
1 public class CLASS
2 {
3   public String generate(Method method, String returnType, String
        businessObjectType, boolean readOnly, Class<?> businessObjectClass,
        String elementType, String packageName, String genClassName, String
        attributeName)
4   {
5     return "";
6   }
7 }
```

Quelltext A.4: Skeleton für Attribut-Mapping-Typen

```
1 package generators;
2
3 import java.lang.reflect.Method;
4
5 public class SimpleMappingGenerator
6 {
7   protected static String nl;
8   public static synchronized SimpleMappingGenerator create(String
        lineSeparator)
9   {
10    nl = lineSeparator;
11    SimpleMappingGenerator result = new SimpleMappingGenerator();
12    nl = null;
13    return result;
14  }
15
16  public final String NL = nl == null ? (System.getProperties().
        getProperty("line.separator")) : nl;
```

```
17  protected final String TEXT_1 = "package ";
18  protected final String TEXT_2 = ";" + NL + "" + NL + "/**" + NL + " *
        Diese Klasse wurde automatisch generiert." + NL + " * Bitte den
        generierten Code nicht editieren." + NL + " * Bitte nur von dieser
        Klasse ableiten." + NL + " *" + NL + " * Attribut-Mapping fuer das
        Business Object: ";
19  protected final String TEXT_3 = NL + " * Attribut: ";
20  protected final String TEXT_4 = NL + " * Attribut-Mapping-Typ:
        SimpleMapping" + NL + " */" + NL + "public class Sm";
21  protected final String TEXT_5 = " extends de.innovations.objectmapping.
        Mapping<";
22  protected final String TEXT_6 = ", ";
23  protected final String TEXT_7 = ">" + NL + "{" + NL + "\tpublic Sm";
24  protected final String TEXT_8 = "(java.lang.String symbolicName)" + NL
        + "\t{" + NL + "\t\tsuper(symbolicName, ";
25  protected final String TEXT_9 = ".class);" + NL + "\t}" + NL + "\t" +
        NL + "\t@Override" + NL + "\tprotected ";
26  protected final String TEXT_10 = " getValue(final ";
27  protected final String TEXT_11 = " businessObject, final de.innovations
        .objectmapping.IMappingContext context) " + NL + "\t{" + NL + "\t\
        treturn businessObject.";
28  protected final String TEXT_12 = "();" + NL + "\t}" + NL + "\t";
29  protected final String TEXT_13 = NL + "\t@Override" + NL + "\tprotected
         void setValue(final ";
30  protected final String TEXT_14 = " ";
31  protected final String TEXT_15 = ", final ";
32  protected final String TEXT_16 = " businessObject, final de.innovations
        .objectmapping.IMappingContext context)" + NL + "\t{" + NL + "\t\
        tbusinessObject.set";
33  protected final String TEXT_17 = "(";
34  protected final String TEXT_18 = ");" + NL + "\t}";
35  protected final String TEXT_19 = NL + "}";
36
37  public String generate(Method method, String returnType, String
        businessObjectType, boolean readOnly, Class<?> businessObjectClass,
        String elementType, String packageName, String genClassName, String
        attributeName)
38  {
39    final StringBuffer stringBuffer = new StringBuffer();
40    stringBuffer.append(TEXT_1);
41    stringBuffer.append(packageName);
42    stringBuffer.append(TEXT_2);
43    stringBuffer.append(businessObjectClass.getSimpleName());
44    stringBuffer.append(TEXT_3);
```

```
45      stringBuffer.append(attributeName);
46      stringBuffer.append(TEXT_4);
47      stringBuffer.append(genClassName);
48      stringBuffer.append(TEXT_5);
49      stringBuffer.append(returnType);
50      stringBuffer.append(TEXT_6);
51      stringBuffer.append(businessObjectClass.getName());
52      stringBuffer.append(TEXT_7);
53      stringBuffer.append(genClassName);
54      stringBuffer.append(TEXT_8);
55      stringBuffer.append(returnType);
56      stringBuffer.append(TEXT_9);
57      stringBuffer.append(returnType);
58      stringBuffer.append(TEXT_10);
59      stringBuffer.append(businessObjectClass.getName());
60      stringBuffer.append(TEXT_11);
61      stringBuffer.append(returnType.contains("Boolean") ? "is" : "get");
62      stringBuffer.append(attributeName);
63      stringBuffer.append(TEXT_12);
64       if(readOnly == false){
65      stringBuffer.append(TEXT_13);
66      stringBuffer.append(returnType);
67      stringBuffer.append(TEXT_14);
68      stringBuffer.append(attributeName.replace(attributeName.substring(0,
           1), attributeName.substring(0, 1).toLowerCase()));
69      stringBuffer.append(TEXT_15);
70      stringBuffer.append(businessObjectClass.getName());
71      stringBuffer.append(TEXT_16);
72      stringBuffer.append(attributeName);
73      stringBuffer.append(TEXT_17);
74      stringBuffer.append(attributeName.replace(attributeName.substring(0,
           1), attributeName.substring(0, 1).toLowerCase()));
75      stringBuffer.append(TEXT_18);
76       }
77      stringBuffer.append(TEXT_19);
78      return stringBuffer.toString();
79    }
80  }
```

Quelltext A.5: Generatorklasse Simple Mapping

Literaturverzeichnis

[AM06] Aniszczyk, Chris ; Marz, Nathan: *Create more – better – code in Eclipse with JET*. http://www.ibm.com/developerworks/opensource/library/os-ecl-jet. Version: 2006. – verfügbar am 05.05.2009

[B04] Bill, Moore ; ...: *Eclipse Development: [using the Graphical Editing Framework and the Eclipse Modeling Framework]*. First Edition. New York : Redbooks, 2004

[Bud03] Budinsky, Frank: *Eclipse modeling framework: A developer's guide*. 2. printing. Boston : Addison-Wesley, 2003 (The eclipse series)

[DAT08] DATACOM Buchverlag GmbH: *Mappings*. http://www.itwissen.info/definition/lexikon/mapping-Abbildung.html. Version: 2008. – verfügbar am 22.10.2008

[DH03] Dunkel, Jürgen ; Holitschke, Andreas: *Softwarearchitektur für die Praxis*. Berlin : Springer, 2003 (Xpert.press)

[E05] Ehrig, Karsten ; ...: *Erstellung eines grafischen Editor-Plug-Ins mit Eclipse EMF und GEF*. http://tfs.cs.tu-berlin.de/publikationen/Papers05/EET05.pdf. Version: 2005. – verfügbar am 07.01.2009

[G02] Gamma, Erich ; ...: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 5., korrigierter Nachdr. München : Addison-Wesley, 2002 (Programmer's choice)

[GB04] Gamma, Erich ; Beck, Kent: *Eclipse erweitern: Prinzipien, Patterns und Plug-Ins*. München : Addison-Wesley, 2004 (Open source library)

[HN05a] Hansen, Robert H. ; Neumann, Gustaf: *UTB für Wissenschaft Uni-Taschenbücher*. Bd. 2669: *Wirtschaftsinformatik 1: Grundlagen und Anwendungen*. 9. Auflage. Stuttgart : Lucius & Lucius, 2005

[HN05b] Hansen, Robert H. ; Neumann, Gustaf: *UTB für Wissenschaft Uni-Taschenbücher*. Bd. 2670: *Wirtschaftsinformatik 2: Informationstechnik*. 9. Auflage. Stuttgart : Lucius & Lucius, 2005

[Inn08] Innovations Software Technology: *work frame relations - Framework zur effizienten Entwicklung individueller Client Management-Frontapplikationen für Banken*. http://www.innovations.de/

fileadmin/pdf/brochure/broschuere_workframerelations.pdf. Version: 2008. – verfügbar am 10.11.2008

[Jen99] Jenz & Partner GmbH: *Views on Technology - Eine Anwendungsarchitektur für adaptive Systeme, Kurzfassung*. http://www.bpiresearch.com/Resources/Download/VOT_Architektur.pdf. Version: 1999. – verfügbar am 28.10.2008

[Krü02] Krüger, Guido: *Handbuch der Java-Programmierung*. 3.Aufl. München : Addison-Wesley, 2002

[Lho06] Lhotka, Rockford: *Expert C# 2005 business objects: [architect, design, and develop highly scalable and maintainable object-oriented business applications]*. 2nd ed. Berkeley, Calif. : Apress, 2006 (Books for professionals by professionals)

[LM07] Louis, Dirk ; Müller, Peter: *Markt & Technik Jetzt lerne ich*. Bd. 24176: *Jetzt lerne ich Java 6: Komplettes Starterkit für den erfolgreichen Einstieg in die Programmierung*. München : Markt-+-Technik-Verl., 2007

[Loc08] Lochbihler, Andreas: *Java Reflection*. http://pp.info.uni-karlsruhe.de/lehre/SS2008/foo/reflection.pdf. Version: 2008. – verfügbar am 17.03.2009

[MSH03] Middendorf, Stefan ; Singer, Reiner ; Heid, Jörn: *Java: Programmierhandbuch und Referenz für die Java-2-Plattform, Standard-Edition*. 3., überarb. und erw. Aufl. Heidelberg : dpunkt-Verl., 2003

[Oes06] Oestereich, Bernd: *Analyse und Design mit UML 2.1: Objektorientierte Softwareentwicklung*. 8., aktualisierte Aufl. München : Oldenbourg, 2006

[Pop04] Popma, Remko: *JET Tutorial Part 1 (Introduction to JET) - Eclipse Corner Article*. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html. Version: 2004. – verfügbar am 23.09.2008

[SS05] Schmauder, Ralf ; Schill, Philipp: *Codegenerierung mit dem „Eclipse Modeling Framework“ und JET*. http://www.sigs.de/publications/os/2005/01/schmauder_schill_OS_01_05.pdf. Version: 2005. – verfügbar am 08.10.2008

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Geringswalde, den 12. Mai 2009

_____

Michel Kleinschmidt